PostgreSQL 中文文档 9.3

wizardforcel

Published with GitBook



目錄

PostgreSQL 中文文档 9.3

来源：PostgreSQL 9.3.1 中文手册

前言

Table of Contents

本书是PostgreSQL的官方文档。它是由PostgreSQL开发人员和其它志愿者撰写的，并且与PostgreSQL软件的开发同步进行。它描述了当前版本的PostgreSQL官方支持的所有功能。

为了便于管理有关PostgreSQL的大量信息，本书被组织成了几个部分。不同的部分针对在PostgreSQL上有着不同经验、不同认识层次、不同阶段的用户：

- Part I是一个给新用户的非正式介绍。
- Part II记载了SQL查询语言环境，包括数据类型和函数以及用户层次的性能调优。每个PostgreSQL用户都应该阅读这些内容。
- Part III描述了服务器的安装和管理。每个运行PostgreSQL的人，不管是个人使用还是为别人维护，都应该阅读这部分内容。
- Part IV描述PostgreSQL客户端程序的编程接口。
- Part V 包含了那些给高级用户查看的信息。其中的内容包括用户定义数据类型和函数等。
- Part VI 包含了有关SQL命令的参考信息以及客户端和服务器程序开发的信息。这部分支持其它部分，其中的信息是结构化组织并且按照命令/程序名排序的。
- Part VII包含可能用于PostgreSQL开发人员的分类信息。

何为PostgreSQL？

PostgreSQL是以加州大学伯克利分校计算机系开发的 POSTGRES, Version 4.2为基础的对象关系型数据库管理系统(ORDBMS)。POSTGRES开创的许多概念在很久以后才出现在商业数据库中。

PostgreSQL是最初伯克利代码的一个开放源码的继承者。它支持大部分SQL标准并且提供了许多其它现代特性：

- 复杂查询
- 外键
- 触发器
- 可更新的视图
- 事务完整性
- 多版本并发控制

另外，PostgreSQL可以用许多方法进行扩展，比如通过增加新的：

- 数据类型
- 函数
- 操作符
- 聚合函数
- 索引方法
- 过程语言

并且，因为许可证的灵活，任何人都可以以任何目的免费使用、修改、分发PostgreSQL，不管是私用、商用、还是学术研究使用。

PostgreSQL简史

现在被称为PostgreSQL的对象-关系型数据库管理系统是从美国加州大学伯克利 分校编写的POSTGRES软件包发展而来的。经过二十几年的发展，PostgreSQL 是目前世界上可以获得的最先进的开放源码数据库系统。

Berkeley 的POSTGRES项目

Michael Stonebraker领导的POSTGRES项目 是由防务高级研究项目局(DARPA)、陆军研究办公室(ARO)、国家科学基金 (NSF)、ESL公司共同赞助的。POSTGRES的实现始于1986年，该系统最初的 概念详见 POSTGRES的设计 ，最早的数据模型定义见 POSTGRES数据模型 。当时的规则系统设计在*POSTGRES*规则系统的设计 里描述。存储管理器的理论基础和体系结构在 POSTGRES存储系统设计 里有详细描述。

从那以后，POSTGRES经历了几次主要的 版本更新。第一个"演示性"系统在1987年便可使用了， 并且在1988年的ACM-SIGMOD大会上展出。在1989年 6月发布了版本1(在 POSTGRES的实现 里有描述)给一些外部的用 户使用。为了回应用户对第一个规则系统的批评 (POSTGRES规则系统注解)，我们重新设计了规则系统 (数据库系统中的规则，过程，缓存和视图)，并在1990年6月发布了使用新规则系统的 版本2。版本3在1991年出现，增加了多存储管理器的支持，并且改进了 查询执行器，重新编写了规则系统。从那以后，随后的版本直到 Postgres95发布前(见下文)，工作都集中 在移植性和可靠性上。

POSTGRES已经在许多研究或实际的应用中 得到了应用。这些应用包括：一个财务数据分析系统、一个喷气引擎 性能监控软件包、一个小行星跟踪数据库、一个医疗信息数据库和一些地理信息系统。POSTGRES还被许多大学用 于教学用途。最后，Illustra Information Technologies(后来并入 Informix ， 而它现在属于 IBM)拿到代码并使之 商业化。在1992年末POSTGRES成为 Sequoia 2000 scientific computing project的首要数据管理器。

到了1993年，外部用户的数量几乎翻番。随着用户的增加，用于源代 码维护的时间日益增加，以至占用了太多本应该用于数据库研究的时 间，为了减少支持的负担，伯克利的POSTGRES 项目在版本4.2时正式终止。

Postgres95

1994年，Andrew Yu和Jolly Chen向POSTGRES 中增加了SQL语言的解释器，并随后将Postgres95 的源代码发布到互联网上供大家使用，从而成为一个开放 源码的原伯克利POSTGRES的继承者。

Postgres95所有源代码都是完全的ANSI C， 而且代码量减少了25%，并且有许多内部修改以利于提高性能和代码的 可维护性。Postgres95版本1.0.x在进行 Wisconsin Benchmark测试时大概比POSTGRES v4.2快 30%-50%。除了修正了一些错误，下面的是一些主要改进：

- 原来的查询语言PostQUEL被SQL取代(在服务器端实现)。 (接口库libpq是以PostQUEL来命名的。) 在PostgreSQL之前还不支持子查询(见下文)，但 这个功能可以在Postgres95里面由用户定义的 SQL函数实现，重新实现了聚集，同时还增加了对 `GROUP BY` 查询子句的支持。
- 新增加了利用GNU Readline进 行交互SQL查询的程序(psql)。这个程序很大程度上取代了 老的monitor程序。
- 增加了新的前端库(`libpgtcl`)，用以支持 以Tcl为基础的客户端。一个样本shell (`pgtclsh`)，提供了新的Tcl命令用于 Tcl程序和Postgres95 后端之间的交互。
- 彻底重写了大对象的接口，将大对象倒转(inversion)作为存储大对象 的唯一机制（去掉了倒转(inversion)文件系统）。
- 去掉了实例级的规则系统，但我们仍然可以通过重写规则来使用规则。
- 在发布的源码中增加了一个简短的常用SQL和 Postgres95特有的SQL特性的教程。
- 用GNU make取代了BSD make用于编译。 Postgres95可以使用不加补丁的 GCC进行编译(修正了偶数字节数据的对齐问题)。

PostgreSQL

到了1996年，我们很明显的看出"Postgres95"这个名字已经经不起 时间的考验了。于是我们起了一个新名字PostgreSQL 用于反映最初的POSTGRES和最新的使用SQL的版本之间的关系。同时版本号也重新从6.0开始，将版本号放回到最 初的由伯克利POSTGRES项目开始的顺序中。

许多人出于习惯或者发音简单的原因，将PostgreSQL 称为"Postgres"（现在很少全部用大写字母），这种称法被当做绰号或者别名而广泛接受。

Postgres95版本的开发重点放在识别和理解服务器端 代码中已有的问题上。PostgreSQL开发重点转到了一些 有争议的特性和功能上面，当然各个方面的工作同时都在进行。

自那以来，PostgreSQL发生的变化可以在Appendix E里面找到。

格式约定

下面的格式用于命令的大纲：方括弧(`[` 和 `]`)表示可选的部分(在Tcl命令里使用的是问号(`?`)。花括弧(`{` 和 `}`) 和竖条(`|`)表示你必须选取一个候选。连续点(`...`) 表示前面的元素可以重复。

如果能提高清晰度，那么SQL命令使用前缀提示符 =>， 而shell命令使用前缀提示符 `$` 。不过，通常是不显示提示符的。

管理员通常是一个负责安装和运行服务器的人。 用户可以是任何使用或者需要使用PostgreSQL系统任何部分的人。我们不应该对这些术 语的概念理解得太狭隘。这份文档集在系统管理步骤方面没有固定的假设。

更多信息

除了文档(也就是本手册)以外，还有一些其他的PostgreSQL相关资源：

Wiki

PostgreSQL wiki包括项目的FAQ (常见问题) 列表, 待办事项 列表,和更多话题的详细信息。

网站

PostgreSQL网站 有最新版本的详细信息以及能让你更高效地使用PostgreSQL的其他信息。

邮件列表

邮件列表是解决用户问题，与其他用户分享经验，与开发者联系的好地方。详见PostgreSQL网站。

你自己!

PostgreSQL是一个开放源码的项目。也就是说， 它依靠用户社区进行持续支持。当你刚开始使用PostgreSQL 时，你将依靠其他人的帮助，或者是通过文档、或者是通过邮件列表。 同时也请将你的知识贡献出来。阅读邮件列表并回答问题。如果你学到了一些文档里没有提到的东西， 请将其写下来并贡献出来。如果你给代码增加了特性，请贡献出来。

臭虫汇报指导

当你在PostgreSQL里碰到臭虫时，我们也希望能知道它。 你的臭虫汇报是将PostgreSQL做得更加可靠的一个非常重要的部分， 因为再细致的工作也不能保证在任何情况、任何平台下PostgreSQL 的每一个部分都能正常工作。

下面的建议试图帮助你正确格式化臭虫报告，这样这些报告就能够以一种有效的方法处理。我们不强迫任何人遵循这些东西，但是这样做对我们每个人都有好处。

我们不能保证能够马上修补每个臭虫。如果臭虫是显而易见的，很关键的或者影响许多用户， 那么很有可能有些人会认真检查它们。同样我们也可能是告诉你升级到一个新版本， 看看臭虫是否仍然存在。否则，我们可能会说这个臭虫在我们正计划的几个主要改写之前不会得到修补。 或者这个臭虫只是太费事了，而且目前的日程表上有更重要的事情要做。如果你立即需要帮助， 那么请考虑获取一个商业性的支持。

标识臭虫

在你报告臭虫之前，请一再仔细地读文档，以确认你确实可以做你在做的事情。 如果文档中对你能否处理你所做的事情并不清楚，也请你汇报过来，因为这个是文档的臭虫。 如果发现你的程序表现的不像文档里说的那样，那就是一个臭虫。 这时可能包括(不过不一定局限于)下面的现像：

- 程序带着一个致命信号或者一个指向程序错误的操作系统错误信息退出。 一个反例是一个"disk full"(磁盘满)信息，因为这样的错误必须在 Postgres 外部进行修复。
- 程序对给出的任何输入都产生错误的输出。
- 程序拒绝接收(文档里定义的那些)有效的输入。
- 程序对非法输入没有生成任何提示或者错误信息。但是需要注意的是， 你认为非法的输入可能是我们设想的扩展或者与传统兼容的做法。
- 在支持的平台上根据指导未能成功地编译或安装PostgreSQL。

这里的"程序"代表任何可执行文件，而不仅仅是后端进程。

速度慢或者资源消耗大不算是臭虫。请阅读文档或者提交邮件列表获取调节你的应用的性能的帮助。 未能遵循SQL标准也不算是一个臭虫，除非文档明确声明了遵守该特定特性。

在你准备继续汇报臭虫之前，请检查 TODO 列表和 FAQ ，看看你报告的臭虫是否已知。 如果你不能解析 TODO 列表里面的信息，请汇报你的问题。至少我们可以把 TODO 列表做得更清晰。

汇报什么

关于汇报臭虫需要记住的最重要的事就是写出所有事实并且只写事实。不要推测你认为是什么错了， 什么"看起来像"，或者是推测程序的哪一部分失灵了。如果你不熟悉 PostgreSQL 的实现， 你很可能猜错因而不能帮我们任何忙。而且即使你熟悉 Postgres 的实现， 提炼出来的解释也只是事实的补充而不是代替。如果我们准备修理这个臭虫， 我们仍然需要首先亲自看到臭虫的出现。报告简单的事实相对而言比较直接 (你可以从屏幕上拷贝和粘贴)， 不过经常发生的是很多人认为这些事实不重要而忽略了重要的细节，否则汇报总是能够被我们理解。

下面的条目应该包含在所有臭虫汇报里面：

- 从程序启动开始到重现问题的准确步骤顺序。这应该自包含的；要知道如果输出将依赖于表中的数据时，光把一个光秃秃的 `SELECT` 语句发过来而不把前面的 `CREATE TABLE` 和 `INSERT` 语句发过来是不够的。我们没有时间分析你的数据库结构，而且如果我们试着建立我们自己的数据， 那我们就有可能错过问题。

  测试与 SQL 语言有关的问题的最好的格式是一个可以通过psql 前端运行并显示问题的文件(确保在 `~/.psqlrc` 启动文件里面没有任何东西)。 一种比较简单的创建这个文件的方法就是用pg_dump 导出表声明和仿真的数据，以及有毛病的查询。我们鼓励你最小化你的例子， 但这不是非做不可的事情。如果臭虫是可以复现的，那么两种方式都能帮助我们找到它。

  如果你的应用使用其它客户端接口，比如PHP，那么请设法隔离出有毛病的查询。 我们可能不会架设一个 web 服务器来复现你的问题。不管怎么说，请记住提供准确的输入文件， 而不要猜测问题会在"大文件"或者"中等尺寸的数据库"等等的身上发生。 因为这样的信息太不确切，因而没有什么用处。

- 你得到的输出。请不要说它"不起作用"或者"失灵了"。如果有错误信息， 请写明，即使你不能理解也一样。如果程序带着操作系统错误退出，也请写清楚。如果什么也没有发生， 就照直说。即使你的测试实例是程序崩溃或者其它显而易见的现像， 它也有可能不会在我们的平台上发生。如果可能，最简单的事情是从终端拷贝输出。

  > **Note:** 如果你报告一个错误信息，请从信息中获取最冗长的版本。在psql里， 事先运行 `\set VERBOSITY verbose` 就行。如果你从服务器日志里抽取信息， 那么就把运行时参数log_error_verbosity设置为 `verbose` ， 这样就会报告所有细节。

  > **Note:** 如果是致命错误，客户端提供的信息可能不会包含所有能得到的信息。这种情况下， 还要看看数据库服务器的输出。如果你没有保留服务器输出，那么现在是做这件事的好机会。

- 还有一样很重要的事是声明你期望的输出。如果你只是写到"这条命令给我这样的输出" 或者"这不是我期望的"，我们可能自己运行它，检查输出， 然后认为看上去很好并且正是我们所期望的输出。我们不应该把时间花在解析你的命令的语义上。 特别是要避免仅

仅说"这不是 SQL 说的"或"这不像 Oracle 做的那样"。 从SQL里挖掘出正确的行为可不是好玩的事情， 我们也不能知道所有其它的关系数据库的特性是怎样的。如果你的问题是程序崩溃， 显然你可以忽略这个条目。

- 任何命令行选项和其它启动选项，包括相关的环境变量或者你从缺省值修改以后的配置文件。 同时，还要准确。如果你使用启动系统时自动启动数据库服务器的预打包的版本， 你应该试着找出这些东西是怎样实现的。

- 任何你做的与安装指导不一致的东西。

- PostgreSQL 版本。你可以运行 `SELECT version();` 命令来检查你正在运行的版本是什么。大多数可执行程序支持 `--version` 选项； 至少 `postgres --version` 和 `psql --version` 应该是可以用的。 如果这个函数或者选项不存在，那我们很可能除了告诉你升级外不会说别的东西。 如果你运行预打包的版本(例如RPM)，请说明，包括那个包可能有的任何子版本号。如果你说的是 Git 快照， 也请说明，包括提交hash。

  如果你的版本比9.3.1低，我们几乎肯定要告诉你升级。在每个新版本里都修补了大量的臭虫， 所以你在老版本的PostgreSQL里碰到的毛病很有可能已经修复掉了。 我们只能对那些使用老版本的PostgreSQL节点提供有限的支持； 如果你要求的比我们能提供的更多，那么考虑一下商业的合同支持。

- 平台信息。这包括内核名称和版本、C 库、处理器、存储器信息。 大多数情况下只需要汇报供应商和版本，但是不要指望每个人都很清楚"Debian" 包括什么东西或者说每个人都运行在i386s上。如果你安装有问题， 那么还要详细汇报你机器上的工具的信息(编译器和 make等)。

不要怕你的臭虫汇报太长，这就是生活。一开始就汇报所有的事情要比让我们从你那里挤出事实要好。 另外，如果你的输入文件非常巨大，先问问有没有人有兴趣查看它也是合理的。 这里有一篇 文章， 概述了一些汇报臭虫的更多的提示。

不要把时间花在寻找如何通过修改输入来消除问题的方法上。 这样很可能不会对解决问题有任何帮助。如果发现不能直接修理臭虫， 你还有时间来查找和共享你的绕过方法。还有，我们再说一次， 不要在猜测臭虫的位置上面浪费时间。我们能够及时找到错误。

在你书写臭虫汇报时，请选用不易混淆的术语。软件包本身被称为"PostgreSQL"， 有时简称为"Postgres"。当你特指后端服务器进程时，请明确说明， 而不要仅仅是说"PostgreSQL 崩溃了"。一个独立后端服务器进程的崩溃和父进程 "postgres"崩溃是相当不同的；如果你是说独立后端进程崩溃了，那么请不要说"服务器崩溃了"， 反之亦然。同样，客户端程序，比如交互式前端"psql" 是和后端完全独立的。请试图说明清楚问题是出现在客户端还是服务器端。

到哪里汇报臭虫

通常，把汇报发到臭虫汇报邮件列表 `<[pgsql-bugs@postgresql.org](mailto:pgsql-bugs@postgresql.org)>` 。 我们要求你为电子邮件消息选用一个描述性的题目，也许就用错误信息的一部分。

另外一个方法是填充 web 表单形式的臭虫报告，你可以在项目的 web 站点找到。 用这种方法输入一个臭虫报告会导致它被发送到 `<[pgsql-bugs@postgresql.org](mailto:pgsql-bugs@postgresql.org)>` 邮件列表。

如果你的臭虫报告隐含带有安全问题，而你不想它立即出现在公开的档案库里， 那么就不要发送到 `pgsql-bugs` 。安全问题可以私下里报告到 `<[security@postgresql.org](mailto:security@postgresql.org)>` 。

不要把臭虫汇报发送到任何用户邮件列表里，例如 SQL 语言邮件列表 `<[pgsql-sql@postgresql.org](mailto:pgsql-sql@postgresql.org)>` 或 `<[pgsql-general@postgresql.org](mailto:pgsql-general@postgresql.org)>` 。这些邮件列表用于回答用户问题， 而且那些订阅者通常不希望接收臭虫汇报。更重要的是，他们很可能不会修理这些臭虫。

还有，请不要向开发者邮件列表 `<[pgsql-hackers@postgresql.org](mailto:pgsql-hackers@postgresql.org)>` 发送臭虫汇报。这个列表用于讨论PostgreSQL的开发， 因而我们很希望能和臭虫汇报分离开。如果修理某个毛病需要更多评论， 我们可能会在这个 `pgsql-hackers` 列表开一个关于你的臭虫的讨论会。

如果你觉得文档有问题，请发电子邮件到文档邮件列表 `<[pgsql-docs@postgresql.org](mailto:pgsql-docs@postgresql.org)>` ， 在你的问题汇报里面指明你认为哪部分有错误。

如果你的臭虫是一个在不支持平台上的移植性问题， 向移植性问题邮件列表 `<[pgsql-hackers@postgresql.org](mailto:pgsql-hackers@postgresql.org)>` 发送电子邮件， 这样我们(还有你)可以一起尝试把PostgreSQL移植到你的平台上。

> **Note:** 由于我们不愿意看到各种各样的垃圾邮件，上面的所有电子邮件地址都是封闭的邮件地址。 也就是说，你需要先申请，然后才能发帖子(用 web 表单提交臭虫报告用不着申请)。 如果你只是想发送邮件而不想接受列表的往来邮件，你可以提交邮件并且把提交选项设置为 `nomail` 。如果需要更多的信息，你可以向 `<[majordomo@postgresql.org](mailto:majordomo@postgresql.org)>` 发送一封邮件， 邮件的正文只有一个单词 `help` 就可以了。

I. 教程

欢迎阅读 PostgreSQL 教程。下面几个章节主要是给那些对于 PostgreSQL关系数据库概念和 SQL 语言尚不熟悉的朋友一次简单介绍。 我们只是假设你有一些关于如何使用计算机的基本知识。并不要求特殊的 Unix 或者编程经验。 这部分的主要目的是给你一些手把手的经验，告诉你一些有关 PostgreSQL 系统的重要方面。我们无意把这部分写成一份涵盖各个主题的完整材料。

在你阅读完这份教程之后你可能想继续阅读 Part II 以获取有关 SQL 语言的更多正规知识，或者是阅读 Part IV获取有关为PostgreSQL开发应用的信息。 那些安装并维护自己的服务器的人还应该看看Part III 。

Table of Contents

Chapter 1. 从头开始

Table of Contents

1.1. 安装

自然，在你想开始使用PostgreSQL之前，你必须安装它。 PostgreSQL很有可能 已经安装到你的节点上了，可能是因为它包含在操作系统的发布里，或者是因为系 统管理员已经安装了它。如果是这样的话，那么你应该从操作系统的文档或者系统 管理员那里获取如何访问PostgreSQL的信息。

如果你不清楚PostgreSQL是否已经安装，或者不知 道你能否用它做自己的实验，那么你可以自己安装。这么做并不难，并且是一次很 好的练习。PostgreSQL可以由任何非特权用户安装， 并不需要超级用户(root)的权限。

如果你准备自己安装PostgreSQL，那么请参考 Chapter 15获取安装的有关信息，安装之后再回到这个 指导手册来。一定要记住要尽可能遵循有关设置合适的环境变量章节里的信息。

如果节点管理员没有按照缺省方式设置各项相关参数，那你还有点额外的活儿要干。 比如，如果数据库服务器机器是一个远程的机器，那你就需要把 `PGHOST` 环境变量设置为数据库服务器那台机器的名字。环境变量 `PGPORT` 也可能 需要设置。最后一招：如果当你试着启动一个应用而该应用报告说不能与数据库建立连 接时，你应该马上与数据库管理员联系，如果你就是管理员，那么你就要参考文档以确 保环境变量被正确的设置了。如果你不理解随后的几段，那么先阅读下一章。

1.2. 体系基本概念

在我们开始讲解之前，我们应该先了解PostgreSQL 系统的基本体系。理解PostgreSQL的组件之间的相 互关系将会使本节显得更清晰一些。

按照数据库术语来说，PostgreSQL使用一种客户端/服务器 的模式。一次PostgreSQL会话由下列相关的进程(程序)组成：

- 一个服务器进程，它管理数据库文件，接受来自客户端应用与数据库的连接，并且代表客 户端在数据库上执行操作。数据库服务器程序叫 `postgres` 。
- 那些需要执行数据库操作的用户的客户端(前端)应用。客户端应用可能本身就是 多种多样的：它们可以是一个字符界面的工具，也可以是一个图形界面的应用， 或者是一个通过访问数据库来显示网页的 web 服务器，或者是一个特殊的数据库 管理工具。一些客户端应用是和PostgreSQL发布一 起提供的，但绝大部分是用户开发的。

和典型的客户端/服务器应用(C/S应用)一样，这些客户端和服务器可以在不同 的主机上。这时它们通过 TCP/IP 网络连接通讯。你应该记住的是，在客户机 上可以访问的文件未必能够在数据库服务器机器上访问(或者只能用不同的文件 名进行访问)。

PostgreSQL服务器可以处理来自客户端的多个并发连接。 因此，它为每个连接启动("forks") 一个新的进程。从这个时候开始，客户端和新服务 器进程就不再经过最初的 `postgres` 进程进行通讯。因此，主服务器总是在运行，等 待客户端连接，而客户端及其相关联的服务器进程则是起起停停。（当然，用户是肯定看不到 这些事情的。我们在这儿谈这些主要是为了完整。）

1.3. 创建一个数据库

看看你能否访问数据库服务器的第一个例子就是试着创建一个数据库。一台运行着 的PostgreSQL服务器可以管理许多数据库。通常我们 会为每个项目和每个用户单独使用一个数据库。

节点管理员可能已经为你创建了可以使用的数据库。他应该已经告诉你这个数据库的 名字。如果这样你就可以省略这一步，并且跳到下一节。

要创建一个新的数据库，在我们这个例子里叫 `mydb`，你可以使用下面的命令：

```
<samp class="literal">$</samp> <kbd class="literal">createdb mydb</kbd>
```

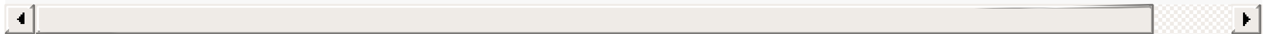
如果不报错那么这一步就成功了，你就可以忽略本节余下的部分了。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

那么就是PostgreSQL没有安装好：要么是就根本没装上、要么是搜索路径没有包含它。 尝试用绝对路径调用该命令试试：

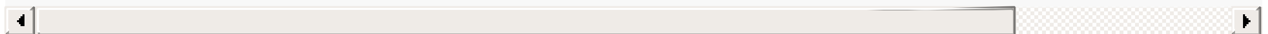
```
<samp class="literal">$</samp> <kbd class="literal">/usr/local/pgsql/bin/createdb mydb</k
```



在你的节点上这个路径可能不一样。请和管理员联系或者看看安装指导以获取正确的位置。

另外一种响应可能是这样：

```
createdb: could not connect to database postgres: could not connect to server: No such fi
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```



这意味着服务器没有启动，或者没有在 `createdb` 预期的地方启动。同样， 你也要检查安装指导或者找管理员。

另外一个响应可能是这样：

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

在这里提到了你自己的登陆名。如果管理员没有为你创建PostgreSQL 用户帐号，就会发生这些现像。PostgreSQL用户帐号和操作系统用户帐 号是不同的。如果你就是管理员，参阅Chapter 20以获取创建用户帐号 的帮助。你需要变成安装PostgreSQL的操作系统用户的身份(通常 是 `postgres`)才能创建第一个用户帐号。也有可能是赋予PostgreSQL 用户名和操作系统用户名不同；这种情况下，你需要使用 `-U` 开关或者使用 `PGUSER` 环境变量声明PostgreSQL用户名。

如果你有个数据库用户帐号，但是没有创建数据库所需要的权限，那么你会看到下面的东西：

```
createdb: database creation failed: ERROR:  permission denied to create database
```

并非所有用户都经过了创建新数据库的授权。如果PostgreSQL 拒绝为你创建数据库，那么你需要让节点管理员赋予你创建数据库的权限。出现这种情况时 请咨询你的节点管理员。如果你自己安装了PostgreSQL，那么 你应该以你启动数据库服务器的用户身份登陆然后参考手册完成权限的赋予工作。 [1]

你还可以用其它名字创建数据库。PostgreSQL允许你在 一个节点上创建任意数量的数据库。数据库名必须是以字母开头并且小于 63 个字节 长。一个方便的做法是创建和你当前用户名同名的数据库。许多工具假设它为缺省的 数据库名，所以这样可以节省敲键。要创建这样的数据库，只需要键入：

```
<samp class="literal">$</samp> <kbd class="literal">createdb</kbd>
```

如果你再也不想使用你的数据库了，那么你可以删除它。比如，如果你是数据库 `mydb` 的属主(创建人)，那么你就可以用下面的命令删除它：

```
<samp class="literal">$</samp> <kbd class="literal">dropdb mydb</kbd>
```

对于这条命令而言，数据库名不是缺省的用户名，你必须明确声明它。这个动作物理 上将所有与该数据库相关的文件都删除并且不可恢复，因此做这件事之前一定要想清楚。

更多关于 `createdb` 和 `dropdb` 的信息可以在createdb和dropdb小节找到。

Notes

[1] 为什么这么干就行了？解释如下：PostgreSQL用户名 是和操作系统用户账号分开的。如果你与一个数据库连接，你可以指定以哪个 PostgreSQL用户名进行连接；如果你不指定，那么缺省 就是你当前的操作系统账号。如果这样，那么总有一个与操作系统用户同名的PostgreSQL用户账号用于启动服务器，并且通常这个用 户都有创建数据库的权限。如果你不想以该用户身份登陆，那么你也可以在任何地方 声明一个 `-U` 选项来选择一个连接时使用的PostgreSQL用户名。

1.4. 访问数据库

一旦创建了数据库，你就可以访问它：

- 运行PostgreSQL交互的终端程序*psql*， 它允许你交互地输入、编辑、执行SQL命令。
- 使用我们现有的图形前端工具，比如pgAdmin或者 带ODBC 或 JDBC支持的办公套件来创建和管理数据库。 这种方法在这份教程中没有介绍。
- 使用多种语言绑定中的一种写一个客户应用。这些可能性在 Part IV中有更深入的讨论。

你可能需要启动 `psql` 来试验本教程中的例子。你可以用 下面的命令为 `mydb` 数据库激活它：

```
<samp class="literal">$</samp> <kbd class="literal">psql mydb</kbd>
```

如果你省略了数据库名字，那么它缺省就是你的用户账号名字。你已经在前面的 使用 `createdb` 小节里了解这一点了。

在 `psql` 里，你会看到下面的欢迎信息：

```
psql (9.3.1)
Type "help" for help.

mydb=>
```

最后一行也可能是

```
mydb=#
```

这个提示符意味着你是数据库超级用户，最可能出现在你自己安装了 PostgreSQL的情况下。作为超级用户意味着你 不受访问控制的限制。对于本教程的目的而言，是否超级用户并不重要。

如果你启动 `psql` 时碰到了问题，那么回到前面的小节。诊断 `createdb` 的方法和诊断 `psql` 的方法很类 似，如果前者能运行那么后者也应该能运行。

`psql` 打印出的最后一行是提示符，它表示 `psql` 正听着你说话，这个时候你就可以敲入SQL查询到一个 `psql` 维护的工作区中。尝试一下下面的命令：

```
<samp class="literal">mydb=></samp> <kbd class="literal">SELECT version();</kbd>
                               version
 -----------------------------------------------------------------------

 PostgreSQL 9.3.1 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

<samp class="literal">mydb=></samp> <kbd class="literal">SELECT current_date;</kbd>
    date
------------
 2002-08-31
(1 row)

<samp class="literal">mydb=></samp> <kbd class="literal">SELECT 2 + 2;</kbd>
 ?column?
----------
        4
(1 row)
```

`psql` 程序有一些不属于 SQL 命令的内部命令。它们以反斜杠 " `\` "开头。比如，你可以用下面的命令获取各 种PostgreSQL SQL命令的帮助语法：

```
<samp class="literal">mydb=></samp> <kbd class="literal">\h</kbd>
```

要退出 `psql` ，键入：

```
<samp class="literal">mydb=></samp> <kbd class="literal">\q</kbd>
```

然后 `psql` 就会退出并且返回到命令行 shell(要获取更多有关 内部命令的信息，你可以在 `psql` 提示符上键入 `\?`)。 `psql` 的完整功能在psql文档中。在这份文档里， 我们将不会明确使用这些特性，但是你自己可以在它们有用的时候使用它们。

Chapter 2. SQL语言

Table of Contents

2.1. 介绍

本章提供一个如何使用SQL执行简单操作的概述。本教程的 目的只是给你一个介绍，并非完整的SQL教程。有许多关于 SQL的书，包括理解新的*SQL*和 *SQL*标准指南。而且你还要知道有些PostgreSQL 语言特性是对标准的扩展。

在随后的例子里，我们假设你已经创建了名为 `mydb` 的数据库， 就像在前面的章里面介绍的一样，并且已经启动了psql。

本手册的例子也可以在PostgreSQL源代码发布目录里的 `src/tutorial/` 中找到。（PostgreSQL 二进制包中可能不编译这些文件）要使用这些文件，先进入该目录然后运行make：

```
<samp class="literal">$</samp> <kbd class="literal">cd `_...._`/src/tutorial</kbd>
<samp class="literal">$</samp> <kbd class="literal">make</kbd>
```

这样就创建了那些脚本以及编译了包含用户定义函数和类型的C文件。如下开始本教程：

```
<samp class="literal">$</samp> <kbd class="literal">cd `_...._`/tutorial</kbd>
<samp class="literal">$</samp> <kbd class="literal">psql -s mydb</kbd>
<samp class="literal">...</samp>

<samp class="literal">mydb=></samp> <kbd class="literal">\i basics.sql</kbd>
```

`\i` 命令从指定的文件中读取命令。 `psql` 的 `-s` 选项把你置于单步模式，它在向服务器发送每个语句之前暂停。本节使用的命令都在 `basics.sql` 文件中。

2.2. 概念

PostgreSQL是一种关系型数据库管理系统 (RDBMS)。这意味着它是一种用于管理那些以关系 形式存储数据的系统。关系实际上是表的数学称呼。今天， 把数据存储在表里的概念已经快成固有的常识了，但是还有其它一些方法用于组织数据库。 在类 Unix 操作系统上的文件和目录就形成了一种层次数据库的例子。更现代的发展是面 向对象的数据库。

每个表都是一个命名的行的集合。每一行由一组相同的命名 字段组成。而且每个字段都有一个特定的类型。虽然每个字 段在每一行里的位置是固定的，但一定要记住 SQL 并未对行在表中的顺序做任何保证 (但你可以对它们进行明确的排序显示)。

表组成数据库，一个由某个PostgreSQL服务器管理的数据库 集合组成一个数据库集群。

2.3. 创建新表

你可以通过声明表的名字和所有字段的名字及其类型来创建表：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

你可以在 `psql` 里连换行符一起键入这些东西。 `psql` 可以识别该命令直到分号才结束。

你可以在 SQL 命令中自由使用空白(空格，tab， 换行符)。这意味着你可以用 和上面不同的对齐方式(甚至在同一行中)键入命令。双划线(" `--` ") 引入注释，任何跟在它后面的东西直到该行的结尾都被忽略。SQL 是对关键字 和标识符大小写不敏感的语言，只有在标识符用双引号包围时才能保留它们的 大小写属性(上面没有这么干)。

`varchar(80)` 声明一个可以存储最长 80 个字符的任意字符串的数据类型。 `int` 是普通的整数类型。 `real` 是一种用于存储单精度浮点数 的类型。 `date` 类型应该可以自解释。没错，类型为 `date` 的字 段名字也是 `date` 。这么做可能比较方便，也可能容易让人 混淆，你自己看啦。

PostgreSQL支持标准的SQL类型： `int` , `smallint` , `real` , `double precision` , `char(``_N_` ), `varchar(``_N_` ), `date` , `time` , `timestamp` ,和 `interval` ， 还支持其它的通用类型和丰富的几何类型。PostgreSQL 允许你自定义任意数量的数据类型。因而类型名并不是语法关键字，除了SQL 标准要求支持的特例外。

第二个例子将保存城市和它们相关的地理位置：

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

`point` 类型就是一个PostgreSQL特有的数据类型的例子。

最后，我们还要提到如果你不再需要某个表，或者你想创建一个不同的表，那么你可以用下面的命令删除它：

```
DROP TABLE _tablename_;
```

2.4. 向表中添加行

`INSERT` 语句用于向表中添加行：

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

请注意所有数据类型都使用了相当明了的输入格式。那些不是简单数字值的常量必 需用单引号(`'`)包围，就像在例子里一样。 `date` 类型实际上 对可接收的格式相当灵活，不过在本教程里，我们应该坚持使用这里显示的格式。

`point` 类型要求一个坐标对作为输入，如下：

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

到目前为止使用的语法要求你记住字段的顺序。一个可选的语法允许你明确地列出字段：

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

如果需要，你可以用另外一个顺序列出字段或者是忽略某些字段，比如说，我们不知道降水量：

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

许多开发人员认为明确列出字段要比依赖隐含的顺序是更好的风格。

请输入上面显示的所有命令，这样你在随后的各节中才有可用的数据。

你还可以使用 `COPY` 从文本文件中装载大量数据。这么干通常更快， 因为 `COPY` 命令就是为这类应用优化的，只是比 `INSERT` 少一些灵活性。比如：

```
COPY weather FROM '/home/user/weather.txt';
```

这里源文件的文件名必须在运行后端进程的那台机器上有效，而不是在客户端上，因为后端进程直接读取这个文件。 你可以在COPY中读到更多有关 `COPY` 命令的信息。

2.5. 查询一个表

要从一个表中检索数据就是查询这个表。SQL 的 `SELECT` 语句就是做这个用途的。该语句分为选择列表(列出要返回的字 段)、表列表(列出从中检索数据的表)、以及可选的条件(声明任意限制)。比如，要检索表 `weather` 的所有行，键入：

```
SELECT * FROM weather;
```

这里的 `*` 是"所有字段"的缩写。 [1] 因此同样的结果可以用下面的语句获得：

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

而输出应该是：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

你可以在选择列表中写任意表达式，而不仅仅是字段列表。比如，你可以：

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

这样应该得到：

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

请注意这里的 `AS` 子句是如何给输出字段重新命名的。 `AS` 子句是可选的。

一个查询可以使用 `WHERE` 子句进行"修饰"，声明需要哪些行。 `WHERE` 子句包含一个布尔表达式（值为真），只有那些布尔表达式为 真的行才会被返回。允许你在条件中使用常用的布尔操作符(`AND` , `OR` , `NOT`)。比如，下面的查询检索旧金山的下 雨天的天气：

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

结果：

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

你可以要求返回的查询是排好序的：

```
SELECT * FROM weather
    ORDER BY city;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

在这个例子里，排序的顺序并非绝对清晰的，因此你可能看到 San Francisco 行 随机的排序。但是如果你使用下面的语句，那么就总是会得到上面的结果：

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

你可以要求查询的结果消除重复行的输出：

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

再次声明，结果行的顺序可能是随机的。你可以组合使用 `DISTINCT` 和 `ORDER BY` 来获取一致的结果： [2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

Notes

[1] 虽然 `SELECT *` 对于即兴的查询是有用的，但我们普遍认为在生产代码中 这是很糟糕的风格，因为给表增加一个字段就改变了结果。

[2] 在一些数据库系统里，包括老版本的PostgreSQL， `DISTINCT` 自动对行进行排序，因此 `ORDER BY` 是多余的。但是这一点并不是 SQL 标准的要求，并且目前的PostgreSQL 并不保证 `DISTINCT` 导致数据行被排序。

2.6. 在表间连接

到目前为止，我们的查询一次只访问了一个表。查询可以一次访问多个表， 或者用某种方式访问一个表，而同时处理该表的多个行。一个同时访问同一个或 者不同表的多个行的查询叫连接查询。举例来说，比如你 想列出所有天气记录以及这些记录相关的城市。要实现这个目标，我们需要拿 `weather` 表每行的 `city` 字段和 `cities` 表所有行的 `name` 字段进行比较，并选取那些这些数值相匹配的行。

> **Note:** 这里只是一个概念上的模型。连接通常以比实际比较每个可能的配对行更高效 的方式执行，但这些是用户看不到的。

这个任务可以用下面的查询来实现：

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

```
     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

观察结果集的两个方面：

- 没有城市 Hayward 的结果行。这是因为在 `cities` 表里面没有与 Hayward 匹配的行，所以连接忽略了 `weather` 表里的不匹配行。我们稍后将 看到如何修补这个问题。
- 有两个字段包含城市名。这是正确的，因为 `weather` 和 `cities` 表的字段是接在一起的。不过，实际上我们不想要这些， 因此你将可能希望明确列出输出字段而不是使用 `*` ：

  ```
  SELECT city, temp_lo, temp_hi, prcp, date, location
      FROM weather, cities
      WHERE city = name;
  ```

练习: 看看省略 `WHERE` 子句的含义是什么。

因为这些字段的名字都不一样，所以分析器自动找出它们属于哪个表，但是如果两个 表中有重复的字段名，你就必须使用字段全称限定你想要的字段：

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

一般认为在连接查询里使用字段全称是很好的风格，这样，即使在将来向其中一个 表里添加了同名字段也不会引起混淆。

到目前为止，这种类型的连接查询也可以用下面这样的形式写出来：

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

这个语法并非像上面那个那么常用，我们在这里写出来是为了让你更容易了解后面 的主题。

现在我们将看看如何能把 Hayward 记录找回来。我们想让查询干的事是扫描 `weather` 表，并且对每一行都找出匹配的 `cities` 表里面的行。如果没有找到匹配的行，那么需要一些"空值"代替 `cities` 表的字段。这种类型的查询叫 外连接 (我们在此之前看到的连接都是内连接)。这样的命令看起来像这样：

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

这个查询是一个左外连接，因为连接操作符(LEFT OUTER JOIN) 左边的表中的行在输出中至少出现一次，而右边的表只输出那些与左边的表中的某些行匹 配的行。如果输出的左表中的行没有右表中的行与其对应，那么右表中的字段将填充为 NULL 。

练习: 还有右连接和全连接。试着找出来它们能干什么。

我们也可以把一个表和它自己连接起来。这叫自连接。 比如，假设我们想找出那些在其它天气记录的温度范围之外的天气记录。 这样我们就需要拿 `weather` 表里每行的 `temp_lo` 和 `temp_hi` 字段与 `weather` 表里其它行的 `temp_lo` 和 `temp_hi` 字段进行比较。我们可以用下面的查询实现这个目标：

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

在这里我们把 weather 表重新标记为 `W1` 和 `W2` 以区分连接的左边和右边。 你还可以用这样的别名在其它查询里节约一些敲键，比如：

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

以后会经常碰到这样的缩写。

2.7. 聚集函数

和大多数其它关系数据库产品一样，PostgreSQL支持聚集函数。 一个聚集函数从多个输入行中计算出一个结果。比如，我们有在一个行集合上计算 `count` (数目)，`sum` (总和)，`avg` (均值)，`max` (最大值)，`min` (最小值)的函数。

比如，我们可以用下面的语句找出所有低温中的最高温度：

```
SELECT max(temp_lo) FROM weather;
```

```
 max
-----
  46
(1 row)
```

如果我们想知道该读数发生在哪个城市，可能会用：

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     _错误_
```

不过这个方法不能运转，因为聚集函数 `max` 不能用于 `WHERE` 子句中。存在这个限制是因为 `WHERE` 子句决定哪些行可以进入聚集阶段；因此它必需在聚集函数之前计算。不过，我们可以用其它方法实现这个目的；这里我们使用子查询：

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```
     city
---------------
 San Francisco
(1 row)
```

这样做是可以的，因为子查询是一次独立的计算，它独立于外层查询计算自己的聚集。

聚集同样也常用于 `GROUP BY` 子句。比如，我们可以获取每个城市低温的最高值：

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

这样每个城市一个输出。每个聚集结果都是在匹配该城市的行上面计算的。 我们可以用 `HAVING` 过滤这些分组：

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

```
  city   | max
---------+-----
 Hayward |  37
(1 row)
```

这样就只给出那些 `temp_lo` 值曾经有低于 40 度的城市。最后， 如果我们只关心那些名字以" `S` "开头的城市，我们可以用：

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'<a name="CO.TUTORIAL-AGG-LIKE">**(1)**</a>
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

(1)

语句中的 `LIKE` 执行模式匹配，在Section 9.7里有解释。

理解聚集和SQL的 `WHERE` 和 `HAVING` 子句之间的关系非常重要。 `WHERE` 和 `HAVING` 的基本区别如下： `WHERE` 在分组和聚集计算之前选取输入行(它控制哪些行进入聚集计算), 而 `HAVING` 在分组和聚集之后选取输出行。因此， `WHERE` 子句不能包含聚集函数；因为试图用聚集函数判断那些行将要输入给聚集运算是没有意义的。 相反， `HAVING` 子句总是包含聚集函数。当然，你可以写不使用聚集的 `HAVING` 子句，但这样做没什么好处，因为同样的条件用在 `WHERE` 阶段会更有效。

在前面的例子里，我们可以在 `WHERE` 里应用城市名称限制，因为它不需要聚集。 这样比在 `HAVING` 里增加限制更加高效，因为我们避免了为那些未通过 `WHERE` 检查的行进行分组和聚集计算。

2.8. 更新

你可以用 `UPDATE` 命令更新现有的行。假设你发现所有 11 月 28 日的温度计数都低了两度，那么你就可以用下面的方式更新数据：

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

看看数据的新状态：

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. 删除

数据行可以用 `DELETE` 命令从表中删除。假设你对 Hayward 的天气不再感兴趣， 那么你可以用下面的命令把那些行从表中删除：

```
DELETE FROM weather WHERE city = 'Hayward';
```

所有属于 Hayward 的天气记录都将被删除。

```
SELECT * FROM weather;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

使用下面形式的语句时一定要小心：

```
DELETE FROM _tablename_;
```

如果没有指定条件， `DELETE` 将从指定表中删除所有行。 做这些之前系统不会请求你确认！

Chapter 3. 高级特性

Table of Contents

3.1. 介绍

在前面几章里，我们介绍了使用SQL存储和访问在PostgreSQL 里的数据的基本方法。我们现在将讨论一些SQL更高级的特性， 这些特性可以简化管理和避免数据的丢失或损坏。最后，我们将看看一些PostgreSQL 的扩展。

本章将不时引用Chapter 2 中的例子，并且对它们进行修改和提高， 因此如果你已经看过那章会更好。本章的一些例子也可以在 tutorial 目录里的 `advanced.sql` 文件里找到。这个文件还包括一些要装载的例子数据，这些数据没有在这里介绍。 请参考Section 2.1获取如何使用该文件的方法。

3.2. 视图

回头看看Section 2.6里的查询。假设你的应用对天气记录和城市位置的 组合列表特别感兴趣，而你又不想每次键入这些查询。那么你可以在这个查询上创建一个视图， 它给这个查询一个名字，你可以像普通表那样引用它。

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

自由地运用视图是好的 SQL 数据库设计的一个关键要素。视图允许我们把表结构的细节封装起来， 这些表可能随你的应用进化而变化，但这些变化却可以隐藏在一个一致的接口后面。

视图几乎可以在一个真正的表可以使用的任何地方使用。在其它视图上面再创建视图也并非罕见。

3.3. 外键

回忆一下Chapter 2里的 `weather` 和 `cities` 表。 考虑下面的问题：你想确保没有人可以在 `weather` 表里插入一条在 `cities` 表里没有匹配记录的数据行。这就叫维护表的参照完整性。在简单的数据库系统里， 实现(如果也叫实现)这个特性的方法通常是先看看 `cities` 表里是否有匹配的记录， 然后插入或者拒绝新的 `weather` 记录。这个方法有许多问题，而且非常不便， 因此PostgreSQL可以为你做这些。

新的表声明看起来会像下面这样：

```
CREATE TABLE cities (
        city     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

然后我们试图插入一条非法的记录：

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

外键的行为可以根据你的应用仔细调节。在这份教程里我们就不再多说了，请你参考Chapter 5 以获取更多的信息。正确使用外键无疑将改进你的数据库应用，所以我们强烈建议你学习它们。

3.4. 事务

事务是所有数据库系统的一个基本概念。一次事务的要点就是把多个步骤捆绑成一个单一的、 不成功则成仁的操作。其它并发的事务是看不到在这些步骤之间的中间状态的，并且如果发生了一些问题， 导致该事务无法完成，那么所有这些步骤都完全不会影响数据库。

比如，假设一个银行的数据库包含各种客户帐户的余额，以及每个分行的总余额。 假设我们要记录一次从 Alice 的帐户到 Bob 的帐户的金额为 $100.00 的支付动作。 那么，完成这个任务的简单到极点的 SQL 命令像下面这样：

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

这些命令的细节在这儿并不重要；重要的是这里牵涉到了好几个独立的更新来完成这个相当简单的操作。 银行官员会希望要么所有这些更新全部生效，要么全部不起作用。我们当然不希望一次系统崩溃就导致 Bob 收到 100 块不是 Alice 支付的钱，也不希望 Alice 老是不花钱从 Bob 那里拿到物品。我们需要保证： 如果在操作的过程中出了差错，那么所有这些步骤都不会发生效果。 把这些更新组合成一个事务就给予我们这样的保证。事务被认为是原子的：从其它事务的角度来看，它要么是全部发生，要么完全不发生。

我们还需要保证：一旦一个事务完成并且得到数据库系统的认可，那么它必须被真正永久地存储， 并且不会在随后的崩溃中消失。比如，如果我们记录到了一个 Bob 撤单的动作， 那么我们不希望仅仅在他走出银行大门之后的一次崩溃就会导致对他的帐户的扣减动作消失。 一个事务型数据库保证一个事务所做的所有更新在事务发出完成响应之前都记录到永久的存储中(也就是磁盘)。

事务型数据库的另外一个重要的性质和原子更新的概念关系密切：当多个事务并发地运行的时候， 每个事务都不应看到其它事务所做的未完成的变化。比如，如果一个事务正忙着计算所有分行的余额总和， 那么它不应该包括来自 Alice 的分行的扣帐和来自 Bob 分行的入帐，反之亦然。所以事务必须是黑白分明的， 不仅仅体现在它们在数据库上产生的永久影响上，而且体现在它们运转时的自身的可视性上。 一个打开的事务所做的更新在它完成之前是无法被其它事务看到的，而到提交的时候所有更新同时可见。

在PostgreSQL里，一个事务是通过把 SQL 命令用 `BEGIN` 和 `COMMIT` 命令包围实现的。因此我们的银行事务实际上看起来像下面这样：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- 等等
COMMIT;
```

如果在该事务的过程中，我们决定不做提交(可能是我们刚发现 Alice 的余额是负数)， 那么我们可以发出 `ROLLBACK` 而不是 `COMMIT` 命令， 那么到目前为止我们的所有更新都会被取消。

PostgreSQL 实际上把每个 SQL 语句当做在一个事务中执行来看待。 如果你没有发出 `BEGIN` 命令，那么每个独立的语句都被一个隐含的 `BEGIN` 和(如果成功的话) `COMMIT` 包围。一组包围在 `BEGIN` 和 `COMMIT` 之间的语句有时候被称做事务块。

> **Note:** 一些客户端库自动发出 `BEGIN` 和 `COMMIT`， 因此你可能不需要特意请求就可以获得事务块的效果。查看你使用的接口的文档。

我们可以通过使用保存点的方法，在一个事务里更加精细地控制其中的语句。 保存点允许你选择性地抛弃事务中的某些部分，而提交剩下的部分。在用 `SAVEPOINT` 定义了一个保存点后，如果需要，你可以使用 `ROLLBACK TO` 回滚到该保存点。 则该事务在定义保存点到 ROLLBACK TO 之间的所有数据库更改都被抛弃， 但是在保存点之前的修改将被保留。

在回滚到一个保存点之后，这个保存点仍然保存着其定义，所以你可以回滚到这个位置好几次。 当然，如果你确信你不需要再次回滚到一个保存点，那么你可以释放它，这样系统可以释放一些资源。 要记住：释放或者回滚到一个保存点都会自动释放在其后定义的所有保存点。

所有这些都发生在一个事务块内部，所以所有这些都不可能被其它事务会话看到。 当且仅当你提交了这个事务块，这些提交了的动作才能以一个单元的方式被其它会话看到， 而回滚的动作完全不会被看到。

记得我们的银行数据库吗? 假设我们从 Alice 的帐户上消费 $100.00，然后给 Bob 的帐户进行加款， 稍后我们发现我们应该给 Wally 的账号加款。那么我们可以像下面这样使用保存点：

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- 呀！加错钱了，应该用 Wally 的账号
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

这个例子当然是实在太简单了，但是通过使用保存点，我们可以对事务块有大量的控制。 并且， `ROLLBACK TO` 是除了事务全部回滚，重新来过之外， 唯一可以用于重新控制一个因错误而被系统置于退出状态事务的方法。

3.5. 窗口函数

窗口函数在和当前行相关的一组表行上执行计算。 这相当于一个可以由聚合函数完成的计算类型。但不同于常规的聚合函数， 使用的窗口函数不会导致行被分组到一个单一的输出行；行保留其独立的身份。 在后台，窗口函数能够访问的不止查询结果的当前行。

这里是一个例子，说明如何比较每个员工的工资和在他或她的部门的平均工资：

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

前三输出列直接来自表 `empsalary`， 并有一个针对表中的每一行的输出行。 第四列将代表所有含有相同的 `depname` 值的表行的平均值作为当前值。 （这实际上与标准 `avg` 聚合函数的功能相同， 但是 `OVER` 子句使其被视为一个窗口函数并在一组合适的行上执行计算。）

窗口函数的调用总是包含一个 `OVER` 子句，后面直接跟着窗口函数的名称和参数。 这是它在语法上区别于普通函数或聚合功能的地方。 `OVER` 子句决定如何将查询的行进行拆分以便给窗口函数处理。 `OVER` 子句内的 `PARTITION BY` 列表指定将行划分成组或分区， 组或分区共享相同的 `PARTITION BY` 表达式的值。 对于每一行，窗口函数在和当前行落在同一个分区的所有行上进行计算。

你还可以使用窗口函数 `OVER` 内的 `ORDER BY` 来控制行的顺序。 （ `ORDER BY` 窗口甚至不需要与行的输出顺序相匹配。）下面是一个例子：

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FR
```

```
  depname  | empno | salary | rank 
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

正如此处所示， `rank` 函数按照由 `ORDER BY` 子句定义的顺序， 在当前行的分区里为每个不同的 `ORDER BY` 值产生了一个数值排名。 `rank` 不需要明确的参数，因为它的行为完全取决于 `OVER` 子句。

窗口函数的行来自查询的 `FROM` 子句产生，并且如果有的话， 经过 `WHERE` , `GROUP BY` 和 `HAVING` 子句过滤的"虚拟表"。 比如，被移除掉的行，因为不符合 `WHERE` 条件，所以是不能被任何窗口函数可见的。 一个查询可以包含多个窗口函数，通过不同的 `OVER` 子句用不同的方式分割数据， 但是他们都作用在这个虚拟表定义的同一个行集合。

我们已经看到了，如果行排序并不重要， `ORDER BY` 可以省略。 在只有一个包含了所有行的分区情况下，也可以省略 `PARTITION BY` 。

还有一个与窗口函数相关的重要的概念：对于每一行，有在其分区范围内的行集， 又称为它的*window frame*。许多（但不是全部）窗口函数，只作用于window frame中的行上， 而不是整个分区。默认情况下，如果使用 `ORDER BY` ， 那么这个frame包含从分区开始到当前行的所有行，以及那些当前行后面的，根据 `ORDER BY` 子句等于当前行的所有行，如果省略 `ORDER BY` ，那么，frame默认包含分区中的所有行。 [1] 下面是一个使用 `sum` 的例子：

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum  
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

如上，因为在`OVER`子句中没有使用`ORDER BY`，因此， window frame与分区(不使用`PARTITION BY`时即整个表)相同；换句话说， 每一次sum求和都是使用表中所有的salary，所以我们得到的每个输出行的结果相同。 但是，如果我们添加`ORDER BY`子句，我们会得到不同的结果：

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

这里的总和是从第一个（最低）工资到当前一个，包括任何当前重复的（注意重复薪金的结果）。

窗口函数仅允许在查询的`SELECT`列表和`ORDER BY`子句中使用。 在其他地方禁止使用，比如`GROUP BY`，`HAVING`和`WHERE`子句。 这是因为它们逻辑上在处理这些子句之后执行。此外，窗口函数在标准聚合函数之后执行。 这意味在一个窗口函数的参数中包含一个标准聚合函数的调用是有效的，但反过来不行。

执行窗口计算后，如果有必要对行进行过滤或分组，你可以使用子查询。例如：

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

上面的查询只显示内部查询结果中`rank`小于3的行。

当查询涉及多个窗口函数时，可以写成每一个都带有单独的`OVER`子句， 但是，如果期待为多个窗口函数采用相同的窗口行为，这样做就会产生重复，并且容易出错。 作为代替，每个窗口行为可以在`WINDOW`子句中进行命名，然后再被`OVER`引用。 例如：

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

有关窗口函数的更多详细信息请查阅Section 4.2.8, Section 9.21,Section 7.2.4,和 SELECT的参考页。

Notes

[1] 当然，还有其他定义window frame的方法，但本教程并不包括它们。详情请参阅Section 4.2.8。

3.6. 继承

继承是面向对象的数据库的概念。它开启了数据库设计的有趣的新的可能性。

让我们创建两个表：一个 `cities` 表和一个 `capitals` 表。 自然，首府(capital)也是城市(cities)，因此在列出所有城市时你想要某种方法隐含地显示首府。 如果你已经很高明了，那么你可能会创造类似下面这样的模式：

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (单位是英尺)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int     -- (单位是英尺)
);

CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

如果只是查询，那么这个方法运转得很好，但是如果你需要更新某几行，那这个方法就很难看了。

一种更好的方法是：

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (单位是英尺)
);

CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

在这个例子里， `capitals` 继承了其父表 `cities` 的所有字段(`name` , `population` 和 `altitude`)。字段 `name` 的类型 `text` 是 PostgreSQL用于变长字符串的固有类型。州首府有一个额外的字段 `state` 显示其所处的州。在PostgreSQL里， 一个表可以从零个或者更多其它表中继承过来。

比如，下面的查询找出所有海拔超过 500 英尺的城市的名字，包括州首府：

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

它返回：

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

另一方面，下面的查询找出所有不是州首府并且位于海拔大于或等于 500 英尺的城市：

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;
```

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

`cities` 前面的 `ONLY` 指示系统只对 `cities` 表运行查询，而不包括继承级别中低于 `cities` 的表。 许多我们已经讨论过的命令— `SELECT` ，`UPDATE` 和 `DELETE` —都支持这个 `ONLY` 表示法。

> **Note:** 尽管继承经常是有用的，但是它还没有集成唯一约束或者外键，因此制约了其实用性。 参阅Section 5.8以获取更多细节。

3.7. 结论

PostgreSQL有许多这份教程里没有谈到的特性， 因为这份教程主要是面向新SQL用户的。这些特性在本书剩余部分将有更详细的介绍。

如果你觉得自己需要更多介绍材料，请访问 PostgreSQL网站获取更多资源的链接。

II. SQL 语言

这一部分描述 PostgreSQL 里面 SQL 语言的使用。我们从描述 SQL 的一般语法开始，然后解释如何创建保存数据的结构、如何填充数据库、以及如何查询数据库。 中间部分列出了用户可以在 SQL 命令中使用的数据类型和函数。 剩下的部分讨论那些对调节数据库、优化其性能很重要的几个方面。

这部分的信息是这样安排的：新手可以从头读到尾，便可以获取相关主题的完整信息， 而不需要向前参考太多次。里面的章节是设计成内容完整独立的，这样高级用户就可以选择独立的章节来阅读。 这部分的信息是按照主题单元以叙述的方式组织的。如果你需要了解特定命令的完整描述，那么应该看看 Part VI。

本书的读者应该知道如何连接 PostgreSQL 数据库并执行 SQL 命令。我们建议那些不熟悉这些方面的读者首先阅读 Part I 。通常 SQL 命令是通过 PostgreSQL 交互终端 psql 输入的，但也可以使用其它有类似功能的程序。

Table of Contents

Chapter 4. SQL语法

Table of Contents

本章描述 SQL 的语法。这些内容是理解随后各章的基础， 那些章里面将详细介绍 SQL 命令如何用于定义和修改数据。

我们也建议那些已经很熟悉 SQL 的用户仔细阅读本章，因为有一些规则和概念在 SQL 数据库之间并不一致，或者是有些东西是PostgreSQL特有的。

4.1. 词法结构

SQL 输入由一系列命令组成。一条命令由一系列记号 构成，用一个分号(";")结尾。输入流的终止也结束一条命令。 哪些记号是合法的取决于特定命令的语法。

记号可以是一个关键字、标识符、 引号包围的标识符、文本(或常量)、 特殊的字符符号。记号通常由空白分隔(空格/tab/换行符)， 但如果不存在混淆的时候也可以不用(通常只是一个特殊字符与一些其它记号类型相连的时候)。

比如，下列命令是(语法上)合法的 SQL 输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这里是三条命令的序列，每条一行(尽管并不要求这么做；多条命令可以在一行里， 单条命令也可以合理地分裂成多行)。

另外，在 SQL 输入里可以有注释。它们不是记号，它们实际上等效于空白。

如果从哪些记号标识命令、哪些是操作数或参数的角度考虑，SQL 语法并不是非常一致。 通常头几个记号是命令名字，因此上面的例子我们通常可以说是一个"SELECT"、 一个"UPDATE"、和一个"INSERT"命令。不过， `UPDATE` 命令总是要求一个 `SET` 在某个位置出现，并且这个特定的 `INSERT` 还要求有一个 `VALUES` 才完整。每条命令的准确语法规则都在Part VI里描述。

4.1.1. 标识符和关键字

像上面例子里的 `SELECT` ， `UPDATE` 或 `VALUES` 这样的记号都是关键字的例子，也就是那些在 SQL 语言里有固定含义的单词。记号 `MY_TABLE` 和 `A` 是标识符的例子。根据使用它们的命令的不同， 它们标识表、字段、或者其它数据库对象的名字。因此，有时候只是简单地叫它们"名字"。关键字和标识符有着同样的词法结构， 意思是我们在没有认识这种语言之前是无法区分一个记号是标识符还是名字。 你可以在Appendix C里找到一个关键字的完整列表。

SQL 标识符和关键字必须以一个字母(`a` - `z` 以及带变音符的字母和非拉丁字母)或下划线(`_`)开头， 随后的字符可以是字母、下划线、数字(`0` - `9`)、 美元符号(`$`)。需要注意的是，根据 SQL 标准，美元符号不允许出现在标识符中， 因此使用美元符号将不易移植。SQL 标准不会定义包含数字或者以下划线开头或结尾的关键字， 因此按照这种格式定义的标识符是安全的，不会和将来标准的扩展特性冲突。

系统使用不超过 `NAMEDATALEN`-1 个字符作为标识符；你可以在命令中写更长的名字，但它们会被截断。 `NAMEDATALEN` 的缺省值是 64， 因此标识符最大长度是 63 字节。如果觉得这个限制有问题，那么你可以在 `src/include/pg_config_manual.h` 里修改 `NAMEDATALEN` 来改变它。

关键字和未被引号包围的标识符都是大小写无关的。因此：

```
UPDATE MY_TABLE SET A = 5;
```

也可以等效地写成：

```
uPDaTE my_TabLE SeT a = 5;
```

一种好习惯是把关键字写成大写，而名字等用小写：

```
UPDATE my_table SET a = 5;
```

还有第二种标识符：分隔标识符或 引号包围的标识符。 它是通过在双引号(`"`)中包围任意字符序列形成的。 分隔标识符总是一个标识符，而不是关键字。因此，你可以用 `"select"` 表示一个字段或者表的名字，而一个没有引号的 `select` 将被当做一条命令的一部分，因此如果把它当做一个表名或者字段名使用的话就会产生一个分析错误。 上面的例子可以用引号包围的标识符这么写：

```
UPDATE "my_table" SET "a" = 5;
```

引号包围的标识符可以包含编码不等于零的任意字符(要包含一个双引号，可以写两个相连的双引号)。 这样我们就可以构造那些原本是不允许的表名或者字段名，比如那些包含空白或与号(&)的名字。 但长度限制依旧。

一个带引号的标识符的变形允许带有代码点标记的逃逸Unicode字符。 该变形以 `U&` 开始（大/小写U后跟有&符号）紧跟着打开的双引号， 之间没有空格，例如 `U&"foo"` 。（需要注意的是， 这可能产生和操作符 `&` 之间的歧义。可以在操作符周围加上空格来避免该问题。）在引号中，通过写一个后面跟有反斜杠和四位十六进制代码点或跟有反斜杠和加号和六位十六进制代码点， Unicode字符可以写成逃逸格式。 例如， `"data"` 可以写成：

```
U&"d\0061t\+000061"
```

下例以西里尔字母写俄文"slon"（象）。

```
U&"\0441\043B\043E\043D"
```

如果需要一个非反斜杠的不同的逃逸，可以通过在字符串之后使用 `UESCAPE` 语句来进行声明，如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

逃逸字符可以是一个十六进制数字以外的任何单个字符，加号，一个单引号，一个双引号，或一个空白字符。需要注意的是，逃逸字符是写在单引号中，而不是双引号中。

为了将逃逸字符写到标识符中，可以将它写两次。

只有服务器字符集是 `UTF8` 时，才会完全使用Unicode逃逸语法。 当使用其他服务器字符集时，只有在ASCII内的（最多 `\007F` ）代码点可以被声明。 4位和6位的数字形式可以被用来将UTF-16代理对声明为大于U+FFFF的带有代码点的字符， 尽管6位数字形式技术的可用性使得这样做没有必要。（代理对不是直接存储的， 首先，它们结合成一个单一的代码点，然后再用UTF-8编码。）

把一个标识符用引号包围起来同时也令它大小写相关，而没有引号包围起来的名字总是转成小写。 比如，我们认为标识符 `FOO` ， `foo` 和 `"foo"` 是等价的PostgreSQL名字，但 `"Foo"` 和 `"FOO"` 与上面三个以及它们之间都是不同的。PostgreSQL里对未加引号的名子总是转换成小写， 这和 SQL 标准是不兼容的，SQL 标准要求未用引号包围起来的名字总是转成大写。因此根据标准， `foo` 等于 `"FOO"` 但不等于 `"foo"` 。 如果你想编写可移植的程序，那么我们建议你要么就总是用引号包围某个名字，要么就从来不引。

4.1.2. 常量

在PostgreSQL里有三种隐含类型的常量： 字符串、位串、数值。常量也可以声明为明确的类型， 这样就可以使用更准确的表现形式以及可以被系统更有效地处理。这些将在后面的小节描述。

4.1.2.1. 字符串常量

SQL 里的一个文本常量是用单引号(`'`)包围的任意字符序列， 比如 `'This is a string'` 。在这种类型的字符串常量里嵌入单引号的标准兼容的做法是敲入两个连续的单引号， 比如 `'Dianne''s horse'` 。注意：两个连续的单引号不是双引号(`"`)。

两个只是通过至少一个换行符的空白分隔的字符串常量会被连接在一起， 并当做它们是写成一个常量处理。比如：

```
SELECT 'foo'
'bar';
```

等效于：

```
SELECT 'foobar';
```

但：

```
SELECT 'foo'      'bar';
```

是非法的语法。这个怪异的行为是SQL声明的，PostgreSQL遵循标准。

4.1.2.2. C风格的逃逸字符串常量

PostgreSQL还允许"逃逸"字符串中的内容， 这是一个PostgreSQL对SQL标准的扩展。逃逸字符串语法是通过在字符串前写字母 `E` (大写或者小写)的方法声明的。比如 `E'foo'` 。（当需要续行包含逃逸字符的字符串时， 仅需要在第一行的开始引号前写上 `E` 就可以了。）在逃逸字符串中， 通过一个反斜杠(`\`)开始C风格的反斜杠逃逸序列，在该逃逸中， 反斜杠与其之后字符的组合代表一个特殊的字节值，可参阅Table 4-1。

Table 4-1. 反斜杠逃逸序列

反斜杠逃逸序列	解释
`\b`	退格
`\f`	进纸
`\n`	换行
`\r`	回车
`\t`	水平制表符
\``_o_ ， \``_oo_ ， \``_ooo_ （ _o_ = 0 - 7)	八进制字节值
`\x``_h_` , `\x``_hh_` ( `_h_` = 0 - 9, A - F)	十六进制字节值
\u``_xxxx_ ， \U``_xxxxxxxx_ （ _x_ = 0 - 9, A - F)	16或32位十六进制Unicode字符值

任何其它跟在反斜杠后面的字符都当做文本看待。因此，要在字符串常量里包含反斜杠， 则写两个反斜杠(`\\`)。另外，PostgreSQL 允许用一个反斜杠来逃逸单引号 `\'` ， 不过，将来版本 的 PostgreSQL 将不允许这么用。所以最好坚持使用符合标准的 `''` 。

你有必要为你所创建的字节序列（特别是在使用八进制或十六进制逃逸时）编写有效的服务器字符集编码字符。 当服务器编码是UTF-8时，应该使用Unicode逃逸或另一种Unicode逃逸语法（参阅Section 4.1.2.3）。 （后者通过写出字节来处理UTF-8字符集，这样做是很繁琐的）。

只有服务器字符集是 `UTF8` 时，才会完全使用Unicode逃逸语法。 当使用其他服务器字符集时，只有在ASCII内的（最多 `\u007F` ）代码点可以被声明。 4位和8位的数字形式可以被用来将UTF-16代理对声明为大于U+FFFF的带有代码点的字符， 尽管8位数字形式技术的可用性使得这样做没有必要。（当服务器编码是 `UTF8` 时使用代理对， 首先，它们结合成一个单一的代码点，然后再用UTF-8编码）

Caution

如果配置参数standard_conforming_strings的值是 `off` ， 那么PostgreSQL将能够识别常规和逃逸字符串常量中的反斜杠逃逸。 然而，在PostgreSQL9.1中，参数值默认为 `on` ，这意味着反斜杠逃逸只能在逃逸字符串常量中识别。 这个行为更为标准兼容，但是可能会使依赖于历史行为的应用程序崩溃，因为历史行为中反斜杠逃逸总是能被识别。 作为一个变通方案，你可以设置这个参数为 `off` ，但是最好是不使用反斜杠逃逸。如果你需要使用反斜杠逃逸来表示特殊的字符， 那么请在字符串常量前加上 `E` 。

除 `standard_conforming_strings` 之外，escape_string_warning 和backslash_quote配置参数也影响字符串常量中反斜杠的处理。 |

编码为零的字符不允许出现在字符串常量中。

4.1.2.3. Unicode逃逸字符串常量

PostgreSQL也支持其他类型的字符串逃逸语法， 允许声明任意的带有代码点标记的Unicode字符。一个逃逸Unicode字符常量以 `U&` 开始（大/小写U后紧跟有&符号）紧跟着打开的单引号， 之间没有空格，例如 `U&'foo'` 。 （这可能产生和操作符 `&` 之间的歧义。可以在操作符周围加上空格来避免该问题。）在引号中， 通过写一个后面跟有四位十六进制代码点或跟有加号和六位十六进制代码点的反斜杠， Unicode字符可以写成逃逸格式。例如， `'data'` 可以写成：

```
U&'d\0061t\+000061'
```

下例以西里尔字母写俄文"slon"（象）。

```
U&'\0441\043B\043E\043D'
```

如果需要一个非反斜杠的不同的逃逸，可以通过在字符串之后使用 `UESCAPE` 语句来进行声明，如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

逃逸字符可以是一个十六进制数字以外的任何单个字符，加号，一个单引号， 一个双引号，或一个空白字符。

只有服务器字符集是 `UTF8` 时，才会完全使用Unicode逃逸语法。 当使用其他服务器字符集时，只有在ASCII内的（最多 `\007F` ）代码点可以被声明。 4位和6位的数字形式可以被用来将UTF-16代理对声明为大于U+FFFF的带有代码点的字符， 尽管6位数字形式技术的可用性使得这样做没有必要。（当服务器编码是 `UTF8` 时使用代理对， 首先，它们结合成一个单一的代码点，然后再用UTF-8编码。）

同样，字符串常量的Unicode逃逸语法只有当配置参数standard_conforming_strings 启用时才能生效。否则，该语法在解析SQL语法时给客户端造成混淆，导致SQL注入或其他安全问题。 如果该参数设为OFF，该语法会带着一条错误消息一起被拒绝。

为了将逃逸字符写到字符串中，可以将它写两次。

4.1.2.4. 美元符引用字符串常量

尽管声明字符串常量的标准方法通常都很方便，但是如果字符串中包含很多单引号或者反斜杠， 那么理解字符串的内容可能就会变得很苦涩，因为每个单引号都要加倍。 为了让这种场合下的查询更具可读性，PostgreSQL允许另外一种称作 "美元符引用"的字符串常量书写办法。一个通过美元符引用声明的字符串常量由一个美元符号 (`$`)、零个或多个字符组成的"标签"、 另一个美元符号、 组成字符串常量的任意字符序列、一个美元符号、与前面相同的标签、一个美元符号组成的。 比如，下面是两个不同的用美元符引用的方法声明"Dianne's horse"的例子：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

请注意，在美元符引用的字符串里， 单引号不允许逃逸。实际上，在一个美元符引用的字符串里， 不允许逃逸任何字符：字符串内容总是按照字面内容书写。反斜杠不是特殊的、 美元符自己也不是特殊的(除非它们和开标签的一部分匹配)。

我们可以通过在不同嵌套级别使用不同的"标签"来实现嵌套。最常见的是写函数定义的时候。比如：

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列 `$q$[\t\r\n\v\\]$q$` 表示一个美元符引用的字符串文本 `[\t\r\n\v\\]` ， 在函数体被PostgreSQL执行的时候，它将被识别出来。 但是因为这个序列不匹配外层的美元符引用分隔符 `$function$` ，所以只要考虑了外层字符串， 它就只是常量里面的普通字符而已。

一个美元符引用字符串的标签(如果有标签的话),遵循和无引号包围的标识符相同的规则， 只是它不能包含美元符。标签是大小写敏感的，因此 `$tag$String content$tag$` 是正确的，而 `$TAG$String content$tag$` 则是错误的。

一个后面紧跟着关键字或者标识符的美元符引用字符串必须用空白与其后的关键字或者标识符隔开； 否则美元符引用分隔符将会被当作标识符的开头部分。

美元符引用不是 SQL 标准，但是在写复杂的字符串文本的时候，它通常比标准的单引号语法更方便。 尤其是在其它常量里表现字符串常量的时候更有用，比如在过程函数定义里。如果用单引号语法， 每个上面例子里的每个反斜杠都必须写四个，它们在作为字符串文本分析的时候会减少为两个， 然后在函数执行的时候在内层字符串常量里会再次被解析为一个。

4.1.2.5. 位串常量

位串常量看起来很像在开引号前面有一个 `B` (大写或小写)的普通字符串 (它们之间没有空白)，比如 `B'1001'` 。位串常量里可以用的字符只有 `0` 和 `1` 。

另外，位串常量可以用十六进制表示法声明，方法是使用前缀 `X` (大写或者小写)， 比如 `X'1FF'` ，其中的每个十六进制位等效于四个二进制位。

两种形式的位串常量都可以像普通字符串常量那样跨行连续。位串常量不能用美元符引用。

4.1.2.6. 数值常量

数值常量接受下列通用的形式：

```
_digits_
_digits_.[`_digits_`][e[+-]`_digits_`]
[`_digits_`]._digits_[e[+-]`_digits_`]
_digits_e[+-]_digits_
```

这里的 `_digits_` 是一个或多个十进制数字(0-9)。 如果有小数点，那么至少有一位在小数点前面或后面。 如果出现了指数标记(`e`)那么至少有一个数字跟在它后面。 在常量里不能有空格或者其它字符。请注意任何前导正号或负号实际上都不认为是常量的一部分； 它是施加于常量的一个操作符。

这里是一些合法的数值常量的例子：

42 3.5 4. .001 5e2 1.925e-3

如果一个数值常量既不包含小数点，也不包含指数， 那么如果它的数值可以放在 `integer` 类型中(32位)，则认为它是 `integer` 类型； 如果它的数值可以放在 `bigint` 中(64位)，则认为它是 `bigint` ， 否则认为它是 `numeric` 类型。包含小数点和/或指数的常量总是被认为是 `numeric` 类型。

给一个数值常量赋予初始数据类型只是类型解析算法的开端。 在大多数情况下该常量会根据环境被自动强制转换成最合适的类型。 必要时，你可以通过强制类型转换把一个数值解析成特定的数据类型。 比如，你可以强制要求把一个数值当作 `real` (`float4`)类型来看，方法是这么写：

```
REAL '1.23'  -- 字符串风格
1.23::REAL   -- PostgreSQL (历史的) 风格
```

这些实际上只是下面讨论的通用转换的特例。

4.1.2.7. 其它类型的常量

任意类型的常量都可以用下列表示法中的任何一种来输入：

```
_type_ '_string_'
'_string_'::_type_
CAST ( '_string_' AS _type_ )
```

其中字符串常量的文本将会被代入到类型 `_type_` 的输入转换过程。其结果是一个该类型的常量。 如果不存在该常量所属类型的歧义，那么可以省略明确的类型转换(比如，当你把它直接赋予一个表字段的时候)， 这种情况下它会自动转换。

其中的字符串常量可以用普通 SQL 表示法或者美元符引用来书写。

我们还可以用函数风格的语法来声明类型转换：

```
_typename_ ( '_string_' )
```

不过并非所有类型名都可以这样使用；参阅Section 4.2.9获取细节。

`::`，`CAST()` 和函数调用语法也可以用于声明任意表达式的运行时类型转换(如Section 4.2.9中讨论的那样)。为了避免语法歧义，`_type_` '`_string_`' 的形式只能用于声明一个简单的字面常量的类型。`_type_` '`_string_`' 的另外一个限制是它不能用于数组类型(要用 `::` 或 `CAST()` 声明一个数组常量的类型)。

`CAST()` 语法遵循 SQL 标准。`_type_` '`_string_`' 语法是标准的一个推广：SQL 只是给少数几种数据类型声明了这个语法，但PostgreSQL 允许将其用于所有类型。 `::` 和函数调用的语法是PostgreSQL的历史用法。

4.1.3. 操作符

一个操作符是最多 `NAMEDATALEN` -1 个(缺省63个)下列字符的序列：

-
  -
    - / < > = ~ ! @ # % ^ & | ` ?

不过，有几个限制：

- `--` 和 `/*` 不能出现在操作符中的任何地方， 因为它们会被当做注释开始对待。
- 多字符操作符不能以 `+` 或 `-` 结束，除非其中至少还包含下列操作符之一：

~ ! @ # % ^ & | ` ?

比如， `@-` 是允许的操作符，但 `*-` 不是。 这个限制允许PostgreSQL在不要求记号之间有空白的情况下分析 SQL 兼容的查询。

当你使用非 SQL 标准的操作符的时候，你通常需要用空白分隔相邻的操作符以避免歧义。 比如，如果你定义了一个叫 `@` 的左单目操作符，那么你就不能写成 `X*@Y` ；而是要写成 `X* @Y` 以确保PostgreSQL 把它读成两个操作符，而不是一个。

4.1.4. 特殊字符

有些非字母数字字符有一些特殊含义，因此不能用做操作符。 它们的用法细节可以在相应的描述语法元素的地方找到。 本节只是描述它们的存在和概括一下这些字符的目的。

- 美元符号(`$`)后面跟着数字用于在一个函数体定义或者预备语句中表示参数的位置。 在其它环境里美元符号可能是一个标识符名字或者是一个美元符引用的字符串常量的一部分。
- 圆括弧(`()`)用于分组和强制优先级的时候含义与平常一样。 有些场合里圆括弧是作为一个特定 SQL 命令的固定语法的一部分要求的。
- 方括弧(`[]`)用于选取数组元素。参阅Section 8.15获取更多信息。
- 逗号(`,`)在一些语法构造里用于分隔一个列表的元素。
- 分号(`;`)结束一条 SQL 命令。它不能出现在一条命令里的任何地方， 除了在引号包围的字符串常量或者标识符中。
- 冒号(`:`)用于从数组中选取"片段"(参阅 Section 8.15)。在一些 SQL 方言里(比如嵌入SQL)，冒号用于前缀变量名。
- 星号(`*`)在某些环境里表示一个表的全部字段或者一个复合类型的值。 在用作聚集函数的参数时还表示该聚集并不需要明确的参数。
- 句点(`.`)用在数字常量里，并用于分隔模式、表、字段名。

4.1.5. 注释

注释是任意以双划线开头并延伸到行尾的任意字符序列，比如：

```
-- 这是标准的 SQL 注释
```

另外，还可以使用C风格的块注释：

```
/* 多行注释
 * 可以嵌套: /* 被嵌套的块注释 */
 */
```

这里注释以 `/*` 开头并扩展到对应的 `*/` 。 这些块注释可以嵌套，就像 SQL标准里说的那样 (但和 C 不一样)， 因此我们可以注释掉一大块已经包含块注释的代码。

注释在进一步的语法分析之前被从输入流中删除并用空白代替。

4.1.6. 操作符优先级

Table 4-2显示了PostgreSQL 里面的操作符的优先级和关联性。大多数操作符都有相同的优先级并且都是左关联的。 操作符的优先级和关联性是硬连接到解析器的。这种情况可能会有不那么直观的行为； 比如，布尔操作符 `<` 和 `>` 与布尔操作符 `<=` 和 `>=` 之间有着不同的优先级。同样，当你把双目和单目操作符组合使用的时候， 有时候也需要加圆括弧。 比如：

```
SELECT 5 ! - 6;
```

会被分析成：

```
SELECT 5 ! (- 6);
```

因为解析器不知道 `!` 被定义成了后缀操作符，而不是中缀操作符 (知道的时候只能是太晚了)。要在本例中获得你需要的特性，你要写成：

```
SELECT (5 !) - 6;
```

这是我们为扩展性付出的代价。

Table 4-2. 操作符优先级(递减)

操作符/元素	关联性	描述
`.`	左	表/字段名分隔符
`::`	左	PostgreSQL特有的类型转换操作符
`[` `]`	左	数组元素选择
`+` `-`	右	单目正号，单目负号
`^`	左	幂
`*` `/` `%`	左	乘，除，模
`+` `-`	左	加，减
`IS`	`IS TRUE` , `IS FALSE` , `IS NULL` , etc	
`ISNULL`	测试是否为NULL	
`NOTNULL`	测试是否不为NULL	
(任何其他的)	左	所有其他的本地和用户定义操作符
`IN`	集合成员	
`BETWEEN`	范围包含	
`OVERLAPS`	时间间隔重叠	
`LIKE` `ILIKE` `SIMILAR`	字符串模式匹配	
`<` `>`	小于，大于	
`=`	右	等于，赋值
`NOT`	右	逻辑非
`AND`	左	逻辑与
`OR`	左	逻辑或

请注意操作符优先级也适用于和上面提到的内置操作符同名的用户定义操作符。 比如，如果你为一些客户数据类型定义一个"+"操作符， 那么它和内置的"+"操作符有同样的优先级，不管用它来干什么。

如果在 `OPERATOR` 语法里使用了模式修饰的操作符名，比如：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

那么 `OPERATOR` 构造就会有Table 4-2 里面为"任何其它的" 操作符显示的缺省优先级。不管什么特定的操作符出现在 `OPERATOR()` 里都是这样。

4.2. 值表达式

值表达式用在各种语法环境中，比如在 `SELECT` 命令的目标列表中， 在 `INSERT` 或 `UPDATE` 中用作新的列值， 或者在许多命令的搜索条件中使用。我们有时候把值表达式的结果叫做标量， 以便与一个表表达式的结果相区别(是一个表)。因此值表达式也叫做 标量表达式(或简称表达式)。 表达式语法允许对来自基本部分的数值进行算术、逻辑、集合、和其它运算。

值表达式是下列内容之一：

- 一个常量或者字面值
- 一个字段引用
- 一个位置参数引用(在函数声明体中或预编写的语句中)
- 一个下标表达式
- 一个字段选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚集表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 一个在圆括弧里面的值表达式(可用于子表达式分组和覆盖优先级)。

除了这个列表以外，还有许多构造可以归类为表达式，但是不遵循任何通用的语法规则。 它们通常有函数或操作符的语义，并且在Chapter 9里合适的位置描述。 一个例子是 `IS NULL` 子句。

我们已经在Section 4.1.2里讨论过常量了。下面的节讨论剩下的选项。

4.2.1. 字段引用

一个字段可以用下面的形式引用：

```
_correlation_._columnname_
```

`_correlation_` 是一个表的名字(可能有模式修饰)， 或者是用 `FROM` 子句这样的方法定义的表的别名。 如果在当前查询所使用的所有表中，该字段名字是唯一的， 那么这个相关名字(correlation)和分隔用的点就可以省略(参见Chapter 7)。

4.2.2. 位置参数

位置参数引用用于标识从外部给 SQL 语句的参数。参数用于 SQL 函数定义语句和预编写的查询。有些客户端库还支持在 SQL 命令字符串外边声明数据值， 这种情况下参数用于引用 SQL 字符串行外的数据。一个参数的形式如下：

```
$_number_
```

比如下面这个 `dept` 函数的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

在函数被调用的时候这里的 `$1` 将引用第一个参数。

4.2.3. 下标

如果一个表达式生成一个数组类型的数值， 那么我们可以通过下面这样的表达式来提取数组中的元素

```
_expression_[_subscript_]
```

或者如果是多个相邻的元素("数组片断")可以用下面的方法抽取

```
_expression_[_lower_subscript_:_upper_subscript_]
```

(这里的方括号 `[ ]` 按照字面文本的方式出现。) 每个 `_subscript_` 自己都是一个表达式，它必须生成一个整数值。

通常，数组 `_expression_` 必须用圆括弧包围， 但如果只是一个字段引用或者一个位置参数，那么圆括弧可以省略。同样， 如果源数组是多维的，那么多个下标可以连接在一起。比如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个例子里的圆括弧是必须的。参阅Section 8.15获取有关数组的更多信息。

4.2.4. 字段选择

如果一个表达式生成一个复合类型(行类型)，那么用下面的方法可以抽取一个指定的字段

```
_expression_._fieldname_
```

通常，行 `_expression_` 必须用圆括弧包围， 但是如果要选取的表达式只是一个表引用或者位置参数，可以省略圆括弧。比如:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

因此，一个全称的字段引用实际上只是一个字段选择语法的特例。 一个重要的特殊情形是提取的表列是一个复合型的字段：

```
(compositecol).somefield
(mytable.compositecol).somefield
```

在这里，括号是必须的，用来指出 `compositecol` 是列名而不是表名， `mytable` 是表名而不是模式名。

在一个选择列表中（查看Section 7.3）， 你可以通过使用 `.*` 来要求所有的组合值字段。

```
(compositecol).*
```

4.2.5. 操作符调用

操作符调用有三种语法：

`_expression_` `_operator_` `_expression_` (双目中缀操作符)
`_operator_` `_expression_` (单目前缀操作符)
`_expression_` `_operator_` (单目后缀操作符)

这里的 `_operator_` 记号遵循Section 4.1.3的语法规则， 或者是记号 `AND` ， `OR` ， `NOT` 之一。 或者是一个被修饰的操作符名：

```
OPERATOR(_schema_._operatorname_)
```

具体存在哪个操作符以及它们是单目还是双目取决于系统或用户定义了什么操作符。 Chapter 9描述了内置的操作符。

4.2.6. 函数调用

函数调用的语法是合法函数名(可能有模式名修饰)，后面跟着包含参数列表的圆括弧：

```
_function_name_ ([`_expression_` [, `_expression_` ... ]] )
```

比如，下面的代码计算 2 的平方根：

```
sqrt(2)
```

内置函数的列表在Chapter 9里。其它函数可由用户添加。

可选的可附加名字的参数，详细请参阅Section 4.3。

> **Note:** 一个接受一个复合类型参数的函数，可以使用字段选择语法调用，相反的， 字段选择可以用函数的风格写出来。也就是说，符号 `col(table)` 和 `table.col` 是可以互换的。这个行为不是SQL标准，但是由 PostgreSQL提供，因为它允许函数使用仿真"计算域"。 获取更多信息，请参阅Section 35.4.3。

4.2.7. 聚集表达式

一个聚集表达式代表一个聚集函数对查询选出的行的处理。 一个聚集函数把多个输入缩减为一个输出值，比如给输入求和或求平均。 一个聚集表达式的语法是下列之一：

```
_aggregate_name_ (_expression_ [ , ... ] [ _order_by_clause_ ] )
_aggregate_name_ (ALL _expression_ [ , ... ] [ _order_by_clause_ ] )
_aggregate_name_ (DISTINCT _expression_ [ , ... ] [ _order_by_clause_ ] )
_aggregate_name_ ( * )
```

这里的 `_aggregate_name_` 是前面定义的聚集(可能是带有模式的全称)， 而 `_expression_` 是一个本身不包含聚集表达式或一个窗口函数调用的任意值表达式。 `_order_by_clause_` 是 `ORDER BY` 子句的一个选项，下面会有描述。

第一种形式的聚集表达式为每个输入行调用聚集。 第二种形式与第一种等价(因为 `ALL` 是缺省值)。 第三种形式为每个表达式中不同的值调用聚集（或者为多个表达式不同的值的集合）。 最后一种形式为每个输入行调用一次聚集，因为没有声明特定的输入值， 通常它只用于 `count(*)` 聚集函数。

大多数的聚集函数忽略了NULL输入，因此在一个或多个表达式中产生NULL的行会被丢弃。 对所有的内置聚集函数而言，这样做是可以的，除非另行说明。

比如， `count(*)` 生成输入行的总数； `count(f1)` 生成 `f1` 不为 NULL 的输入行数，因为 `count` 忽略NULL； `count(distinct f1)` 生成 `f1` 唯一且非 NULL 的行数。

一般情况下，输入行会以非特定顺序放入到聚集函数中。在许多情况下，这样做是没有影响的； 如，无论以什么顺序输入， `min` 输出相同的结果。然而， 一些聚集函数（如 `array_agg` 和 `string_agg` ）并非如此。 当使用这种聚集函数时，可以用 `_order_by_clause_` 选项指定输入的顺序。 除了它的表达式仅仅只是表达式，并且不能输出列名或列数之外， `_order_by_clause_` 与 `ORDER BY` 查询子句有相同的语法结构，在Section 7.5中有描述，如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚集函数时需要注意， `ORDER BY` 子句要在所有的聚集函数参数之后，如这样写：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不是：

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

后者在语法上是有效的，但它表示的是， 有两个 `ORDER BY` 关键字的单参数的聚集函数的调用（第二个是无用的， 因为它是一个常量）。

如果 `_order_by_clause_` 中声明了 `DISTINCT` ， 那么所有的 `ORDER BY` 表达式必须匹配规则的聚集参数，也就是说， 不能对没有包含在 `DISTINCT` 列表中的表达式进行排序。

> **Note:** 同时在一个聚集函数中声明 `DISTINCT` 和 `ORDER BY` 是PostgreSQL的一个扩展。

预定义的聚集函数在Section 9.20里描述。其它聚集函数可以由用户增加。

一个聚集表达式只能在 `SELECT` 命令的结果列表或者 `HAVING` 子句里出现。 禁止在其它子句里出现(比如 `WHERE` 子句)，因为这些子句逻辑上在生成聚集结果之前计算。

如果一个聚集表达式出现在一个子查询里(参阅Section 4.2.11 和Section 9.22)，聚集通常是在子查询中进行计算。 但是如果聚集的参数只包含外层查询的变量则例外：这个聚集会属于离他最近的外层查询， 并且在该查询上进行计算。该聚集表达式整体上属于它出现的子查询对外层查询的引用， 其作用相当于子查询每一次计算中的一个常量。前述限制(聚集表达式只能出现在结果列或者 `HAVING` 子句中)只适用于聚集所属的查询层。

4.2.8. 窗口调用函数

通过查询筛选出的行的某些部分，窗口调用函数实现了类似于聚集函数的功能。 不同的是，窗口调用函数不需要将查询结果打包成一行输出—在查询输出中，每一行都是分开的。 然而，窗口调用函数可以扫描所有的行，根据窗口调用函数的分组规范(`PARTITION BY` 列)， 这些行可能会是当前行所在组的一部分。一个窗口调用函数的语法是下列之一：

```
_function_name_ ([`_expression_` [, `_expression_` ... ]]) OVER ( _window_definition_ )
_function_name_ ([`_expression_` [, `_expression_` ... ]]) OVER _window_name_
_function_name_ ( * ) OVER ( _window_definition_ )
_function_name_ ( * ) OVER _window_name_
```

这里的 `_window_definition_` 具有如下语法：

```
[ _existing_window_name_ ]
[ PARTITION BY _expression_ [, ...] ]
[ ORDER BY _expression_ [ ASC | DESC | USING _operator_ ] [ NULLS { FIRST | LAST } ] [, .
[ _frame_clause_ ]
```

选项 `_frame_clause_` 可以是：

```
[ RANGE | ROWS ] _frame_start_
[ RANGE | ROWS ] BETWEEN _frame_start_ AND _frame_end_
```

`_frame_start_` 和 `_frame_end_` 可以是：

```
UNBOUNDED PRECEDING
_value_ PRECEDING
CURRENT ROW
_value_ FOLLOWING
UNBOUNDED FOLLOWING
```

在这里， `_expression_` 表示的是任何自己不含窗口调用函数的值表达式。

`_window_name_` 引用的是查询语句中 `WINDOW` 子句定义的命名窗口规范。命名窗口规范通常只是用 `OVER` `_window_name_` 来引用，但它也可以在括号里写一个窗口名，并且可以有选择的使用排序和/或框架(frame）子句 （如果应用这些子句的话，那么被引用的窗口必须不能有这

些子句）。 后者语法遵循相同的规则（修改 `WINDOW` 子句中已有的窗口名）。 参阅SELECT 查看更多资料。

`PARTITION BY` 选项将查询的行分为一组进入*partitions*， 这些行在窗口函数中单独处理。 `PARTITION BY` 和查询级别 `GROUP BY` 子句做相似的工作，除了它的表达式只能作为表达式不能作为输出列的名字或数。 没有 `PARTITION BY` ，所有由查询产生的行被视为一个单独的分区。 `ORDER BY` 选项决定分区中的行被窗口函数处理的顺序。它和查询级别 `ORDER BY` 子句做相似的工作， 但是同样的它不能作为输出列的名字或数。没有 `ORDER BY` ，行以一个不被预知的顺序处理。

对这些窗口函数（在这个框架而不是整个分区上的）， `_frame_clause_` 指定构成*window frame*的行， 他们是当前分区的一个子集。框架可以用 `RANGE` 或 `ROWS` 模式声明；不管哪种情况， 它的变化范围是从 `_frame_start_` 到 `_frame_end_` 。如果省略了 `_frame_end_` 默认为 `CURRENT ROW` 。

一个 `_frame_start_` 的 `UNBOUNDED PRECEDING` 意味着框架从分区中的第一行开始， 相似的，一个 `_frame_end_` 的 `UNBOUNDED FOLLOWING` 意味着框架从分区中的最后一行结束。

在 `RANGE` 模式中， `_frame_start_` 的 `CURRENT ROW` 意味着框架从当前行的第一个*peer*行开始（ `ORDER BY` 认为等于当前行的行）， 而 `_frame_end_` 的 `CURRENT ROW` 意味着框架从最后一个同等的行结束。在 `ROWS` 模式中， `CURRENT ROW` 简单的意味着当前行。

`_value_` `PRECEDING` 和 `_value_` `FOLLOWING` 当前只允许 `ROWS` 模式。这也就意味着，框架从当前行之前或之后指定的行数启动或结束。 `_value_` 必须是整型表达式，而不能包含变量，聚集函数，或者窗口函数。 该值不能为空或负，但可以是零，表示只选择当前行本身。

默认的框架选项是 `RANGE UNBOUNDED PRECEDING` ，该选项与 `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 相同。有 `ORDER BY` ， 它设置框架从分区的开始一直到与当前行相同的最后一行。没有 `ORDER BY` ， 那么就是当前分区的所有行都包含在框架中，因为所有行都会成为当前行的相同行。

限制条件是 `_frame_start_` 不能为 `UNBOUNDED FOLLOWING` ， `_frame_end_` 不能为 `UNBOUNDED PRECEDING` ，并且 `_frame_end_` 选项不能在上面的列表中出现的比 `_frame_start_` 选项早—例如 `RANGE BETWEEN CURRENT ROW AND` `_value_` PRECEDING是不被允许的。

内置窗口函数在Table 9-48中有描述。其他窗口函数， 用户可以自己添加。同样，任意内置或用户自定义聚集函数可以同窗口函数一样使用。

使用 `*` 的语法可以用来调用无参数的聚集函数为窗口函数，如 `count(*) OVER (PARTITION BY x ORDER BY y)` 。星号(`*`) 通常不用于非聚集的窗口函数。与通常的聚集函数不同，聚集窗口函数不允许在函数参数列中使用 `DISTINCT` 或 `ORDER BY` 。

窗口调用函数只能在 `SELECT` 列，或者查询的 `ORDER BY` 子句中使用。

更多关于窗口函数的信息可以参考： Section 3.5, Section 9.21, Section 7.2.4.

4.2.9. 类型转换

一个类型转换声明一个从一种数据类型到另外一种数据类型的转换。 PostgreSQL接受两种等效的类型转换语法：

```
CAST ( _expression_ AS _type_ )
_expression_::_type_
```

`CAST` 语法遵循 SQL 标准； `::` 语法是PostgreSQL历史用法。

如果对一个已知类型的值表达式应用转换，它代表一个运行时类型转换。 只有在已经定义了合适的类型转换操作的情况下，该转换才能成功。 请注意这一点和用于常量的转换略有区别(如Section 4.1.2.7所示)。 一个应用于字符串文本的转换表示给该字符串文本的常数值赋予一个初始类型， 因此它对于任何类型都会成功(如果字符串文本的内容符合该数据类型的输入语法)。

如果一个值表达式的值对某类型而言不存在混淆的情况，那么我们可以省略明确的类型转换(比如， 在给一个表字段赋值的时候)，而由系统自动执行类型转换。不过， 自动转换只适用于那些系统表中标记着"OK to apply implicitly"的转换函数。 其它转换函数必须用明确的转换语法调用。这些限制是为了避免一些怪异的转换被自动的应用。

我们也可以用函数风格的语法声明一个类型转换：

```
_typename_ ( _expression_ )
```

不过，这个方法只能用于那些类型名同时也是有效函数名的类型。比如， `double precision` 就不能这么用，但是等效的 `float8` 可以。同样， `interval` ， `time` 和 `timestamp` 如果加了双引号也只能这么用，因为存在语法冲突。因此，函数风格的类型转换会导致不一致， 所以应该避免这么使用。

> **Note:** 函数风格语法实际上就是一个函数调用。如果使用两种标准转换语法做运行时转换， 那么它将在内部调用一个已注册的函数执行转换。通常， 这种转换函数和它们的输出类型同名， 因此"函数风格语法"只不过是直接调用底层转换函数。 但是可以移植的程序不能依赖这一点。详情请参阅CREATE CAST。

4.2.10. 排序规则表达式

`COLLATE` 子句重写了表达式的排序规则。它附加到要应用的表达式上：

```
_expr_ COLLATE _collation_
```

这里的 `_collation_` 是一个可能的模式限定标识符。 `COLLATE` 子句绑定得比操作符更紧密；需要时可以用括号。

如果没有明确声明排序规则，数据库系统要么从表达式中的列获取一个排序规则， 要么如果表达式中没有包含列，使用数据库的默认排序规则。

`COLLATE` 子句的两个常见的使用是重写 `ORDER BY` 子句里的排序次序， 例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

和重写执行结果区域敏感的函数或运算符调用的排序规则，例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

请注意，在后面一种情况下， `COLLATE` 子句附加到我们希望作用的运算符的输入参数。 `COLLATE` 子句附加到运算符或者调用函数的哪个参数不重要， 因为运算符或者函数的排序规则是考虑所有参数得到的，并且一个明确的 `COLLATE` 子句将重写所有其他参数的排序规则。（附加不匹配的 `COLLATE` 子句到多个参数， 是一个错误。更多详细信息请参阅Section 22.2。）因此， 下面的例子给出前一个例子相同的结果：

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

但是这样做是错误的：

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试应用一个排序规则到 `>` 运算符的结果，而这个结果是非排序规则类型 `boolean` 。

4.2.11. 标量子查询

一个标量子查询是一个放在圆括弧里只返回一行一列的普通 `SELECT` 查询(参阅Chapter 7获取有关书写查询的信息)。该 `SELECT` 将被执行，而其返回值将在周围的值表达式中使用。 把一个返回超过一行或者超过一列的查询用做标量查询是错误的。 （不过，在一个特定的表达式中，子查询不返回行则不算错误；标量结果被认为是 NULL)。 子查询可以引用外围查询的变量，这些变量在每次子查询中当做常量使用。 参见Section 9.22以获取其它包含子查询的表达式。

比如，下面的查询找出每个州中的最大人口数量的城市：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. 数组构造器

一个数组构造器是一个表达式，它从自身成员元素上构造一个数组值。 一个简单的数组构造器由关键字 `ARRAY` 、一个左方括弧 `[` 、一个或多个表示数组元素值的表达式(用逗号分隔)、一个右方括弧 `]` 组成。比如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

默认的，数组元素类型是成员表达式的公共类型，使用和 `UNION` 或 `CASE` 构造一样的规则决定(参阅Section 10.5)。 你可以通过明确地转换数组构造器为想要的类型来重写这个规则，例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

这和单独构造每个表达式为数组元素类型有相同的效果。关于构造的更多信息，请参阅Section 4.2.9。

多维数组值可以通过嵌套数组构造器的方法来制作。内层构造器中的 `ARRAY` 关键字可以省略。比如，下面的两句生成同样的结果：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

因为多维数组必须是方形，所以同层的内层构造器必须生成同维的子数组。 任何应用于外层 `ARRAY` 构造器的类型转换自动的应用到所有的内层构造器。

多维数组构造器元素可以是任何生成合适数组的东西，而不仅仅是一个子 `ARRAY` 构造。比如：

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

因为数组必须得有类型，因此在构造一个空数组时，必须明确的将其构造成需要的类型，如：

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

我们也可以从一个子查询的结果中构造一个数组。此时， 数组构造器是关键字 `ARRAY` 后跟着一个用圆括弧(不是方括弧)包围的子查询。比如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                                 array
-----------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)
```

子查询必须只返回一个单独的字段。生成的一维数组将为子查询里每行结果生成一个元素，元素类型匹配子查询的输出字段。

用 `ARRAY` 建立的数组下标总是从1开始。 有关数组的更多信息，参阅Section 8.15。

4.2.13. 行构造器

行构造器是一个从提供给它的成员字段数值中构造行值(也叫复合类型值)的表达式。 一个行构造器由关键字 `ROW` 、一个左圆括弧、零个或多个作为行字段值的表达式(用逗号分隔)、 一个右圆括弧组成。比如：

```
SELECT ROW(1,2.5,'this is a test');
```

如果在列表里有多个表达式，那么关键字 `ROW` 是可选的。

行构造器可以包含 `_rowvalue_``.*` 语法， 它将被扩展为行值元素的列表，就像将 `.*` 语法用于一个 `SELECT` 列表顶层一样。例如，如果表 `t` 有 `f1` 和 `f2` 两个字段， 那么下面两句是等价的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note:** 在PostgreSQL 8.2之前， `.*` 语法是不会被扩展的， 所以 `ROW(t.*, 42)` 将创建一个两字段的行，其第一个字段是另一行的值。 新的行为通常更有用。如果你需要旧式的嵌套行值的做法，请将内部的行值写成不包含 `.*` ， 比如 `ROW(t, 42)` 。

缺省时， `ROW` 表达式创建的值是一个匿名的记录类型。如果必要， 你可以把它转换成一个命名的复合类型(既可以是一个表的行类型， 也可以是一个用 `CREATE TYPE AS` 创建的复合类型)。 可能会需要一个明确的转换以避免歧义。比如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 因为只有一个getf1()存在，所以不需要类型转换
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 现在我们需要类型转换以表明调用哪个函数：
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

行构造器可以用于制作存储在复合类型字段中的复合类型值， 或者是传递给一个接受复合类型参数的函数。另外， 我们也可以用它比较两个行值或者用 `IS NULL` 或 `IS NOT NULL` 测试一个行值，比如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

更多的细节，请参阅Section 9.23。 行构造器还可以用于连接子查询，这些在Section 9.22里面有详细讨论。

4.2.14. 表达式计算规则

子表达式的计算顺序是未定义的。特别要指出的是， 一个操作符或者函数的输入并不一定是按照从左向右的顺序或者以某种特定的顺序进行计算的。

另外，如果一个表达式的结果可以通过只判断它的一部分就可以得到， 那么其它子表达式就可以完全不计算了。比如，如果我们这么写：

```
SELECT true OR somefunc();
```

那么 `somefunc()` 就(可能)根本不会被调用。 即使像下面这样写也是一样：

```
SELECT somefunc() OR true;
```

请注意这和某些编程语言里从左向右"短路"布尔操作符是不一样的。

因此，拿有副作用的函数作为复杂表达式的一部分是不明智的。 在 `WHERE` 和 `HAVING` 子句里依赖副作用或者是计算顺序是特别危险的， 因为这些子句都是作为生成一个执行规划的一部分进行了大量的再处理。 在这些子句里的布尔表达式(`AND` / `OR` / `NOT` 的组合)可以用布尔代数运算律允许的任何方式进行识别。

如果需要强制计算顺序，那么可以使用 `CASE` 构造(参阅Section 9.17)。 比如，下面是一种企图避免在 `WHERE` 子句里被零除的不可靠方法：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

但是下面这个是安全的：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

这种风格的 `CASE` 构造会阻止优化，因此应该只在必要的时候才使用。 在这个特殊的例子里，毫无疑问写成 `y > 1.5*x` 更好。

4.3. 调用函数

PostgreSQL允许函数有命名参数，可以被位置 或名称表示法调用。名称表示法对有大量参数的函数特别有用， 因为它更加明确和可靠的标记了形参和实参之间的联系。 在位置表示法里，一个函数调用的参数值要用与函数声明相同的顺序来写出。 在名称表示法里，参数是通过名称来与函数参数相匹配的，可以以任意顺序写出。

不管用那种表示法，在函数声明时给出的有默认值的参数在调用时不必写出。 但是这在名称表示法中是特别有用的，因为参数的任意组合都是可以省略的。 而在位置表示法中，参数只能从右到左省略。

PostgreSQL也支持混合表示法， 混合表示法结合了位置和名称表示法。因为这个原因，先写位置参数然后跟着写命名参数。

下面的例子将说明三种表示法的用法，使用下面的函数定义：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数 `concat_lower_or_upper` 有两个强制的参数， `a` 和 `b` 。 此外第三个参数是一个可选参数 `uppercase` ，默认为 `false` 。 `a` 和 `b` 输入将被串联，并且将根据 `uppercase` 参数强制为大写或者小写。这个函数定义的其他详细资料在这并不重要（参阅Chapter 35 获取更多信息）。

4.3.1. 使用位置表示法

在PostgreSQL中，位置表示法是传递参数到函数的传统机制。 一个例子是：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

所有的参数都按顺序指定。因为 `uppercase` 被指定为 `true` ， 所以结果为大写。另外一个例子是：

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

这里，省略了参数 `uppercase` ，所以接受它的默认值 `false` ， 导致小写的输出。在位置表示法中，参数只要有默认值就可以从右到左省略。

4.3.2. 使用名称表示法

在名称表示法中，每个参数名字是使用 `:=` 声明的，用来将它从参数表达式中独立出来。 例如：

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

另外，参数 `uppercase` 是省略的，所以它被隐式的设置为 `false` 。 使用名称表示法的好处之一是参数可以用任意顺序声明，例如：

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World', uppercase := true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. 使用混合表示法

混合表示法结合了位置和名称表示法。然而，就像之前提到的，命名参数不可以在位置参数前面。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase := true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

在上面的查询中，参数 `a` 和 `uppercase` 是用位置声明的，而 `uppercase` 是用名称声明的。在这个例子中，添加了文档中没有的一点。 在一个有多个参数有默认值的更复杂的函数中，名称或者混合表示法可以节省很多敲键， 并且可以减少犯错的几率。

Chapter 5. 数据定义

Table of Contents

本章介绍如何创建一个保存数据的数据库结构。在关系型数据库里，裸数据是存储在表中的， 因此本章的大部分内容都将用于介绍如何创建表以及如何修改他们， 以及在控制表中存储的数据上有什么可以获得的特性。随后，我们讨论表是如何组织成模式的， 以及如何给表赋予权限。最后，我们将简单查看一下影响数据存储的其它因素，比如继承、 视图、函数、触发器。

5.1. 表的基本概念

关系型数据库中的表非常类似纸面上的一张表：它由行和列组成。字段的数目和顺序是固定的， 每个字段都有一个名字。行的数目是变化的(它反映了给定时刻存储的数据量)。 SQL 对表中行的顺序没有任何承诺。当读取一个表时，行将会以一个未指定的顺序出现， 除非你明确地要求排序。这些内容在Chapter 7里介绍。 另外，SQL 并不给行赋予唯一的标识，因此我们很可能在一个表中有好几个完全相同的行。 这是作为 SQL 基础的下层数学模型的必然结果，但是这通常是我们不愿意看到的。 本章稍后的部分将讨论如何处理这个问题。

每个字段都有一个数据类型。数据类型控制着一个字段所有可能值的集合， 并且控制着字段中数据的语义，这样它就可以用于计算。比如， 一个声明为数值类型的字段不会接受任意文本字符串，而存储在这种字段里的数据可以用于数学计算。 相比之下，一个声明为字符串类型的字段接受几乎任意类型的数据， 但是它们不能进行数学计算(不过可以进行像字符串连接之类的操作)。

PostgreSQL包含一套可剪裁的内置数据类型， 这些类型可以适用于许多应用。用户也可以定义它们自己的数据类型。 大多数内置的数据类型有显而易见的名字和语义， 因此我们把详细的解释放在了Chapter 8。常用的数据类型有： 用于整数的 `integer` 、用于可能为分数的 `numeric` 、 用于字符串的 `text` 、用于日期的 `date` 、用于时间的 `time` 、 用于时间戳的 `timestamp` 。

要创建一个表，可用使用CREATE TABLE命令。在这个命令里， 你至少要为新表声明一个名字，还有各字段的名字以及其数据类型。比如：

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

这样就创建了一个有两个字段的名为 `my_first_table` 的表。 第一个字段的名字是 `first_column` ，数据类型为 `text` ；第二个字段的名字是 `second_column` ，数据类型是 `integer` 。 表和字段的名字遵循Section 4.1.1里面解释的标识符语法。 类型名通常也是标识符(但是有一些例外)。请注意字段列表是逗号分隔的，并且用圆括弧包围。

当然，前面只是一个非常虚构的例子。通常，你会给表和字段取一个有意义的名字， 用以表达他们存储的什么类型的数据，所以还是让我们给一个比较现实的例子：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

`numeric` 类型可以存储分数部分，金额很可能有这样的分数部分。

> **Tip:** 如果你创建了许多相互关联的表，那么最好为表和字段选择一致的命名模式。 比如，表名字可以统一选择单数或者复数，两种选择都有这样那样的理论家支持。

一个表能包含的字段数目是有限制的。根据字段类型的不同， 这个数目可能在 250 到 1600 之间。不过，不管是哪一端的数字， 如果你设计的表包含那么多的字段好像都很不可能发生，否则是设计上有问题的表现。

如果你不再需要一个表，那么可以用DROP TABLE命令删除它。像这样：

```
DROP TABLE my_first_table;
DROP TABLE products;
```

试图删除一个不存在的表是一个错误。不过，在 SQL 脚本文件里， 我们通常在创建表之前无条件删除它并忽略错误信息，所以无论要删除的表存不存在，这个脚本都成功。 当然你还可以使用 `DROP TABLE IF EXISTS` 来避免警告信息， 不过这并不符合 SQL 标准。

如果你需要修改一个已经存在的表，那么可以看看本章稍后的Section 5.5。

使用到目前为止讨论的工具我们可以创建功能完整的表。 本章剩下的部分是有关向表定义中增加特性、保证数据完整性、安全性或便利性的内容。 如果你急于给表填充数据，那么你可以忽略余下的部分直接到Chapter 6， 然后在稍后的时候再回来阅读本章。

5.2. 缺省值

一个字段可以赋予缺省值。如果新创建了一个数据行，而有些字段的数值没有声明， 那么这些字段将被填充为它们各自的缺省值。 一条数据修改命令也可以明确地要求把一个字段设置为它的缺省值，而不用事先知道这个缺省值是什么。 有关数据操作的命令在Chapter 6。

如果没有明确声明缺省值，那么缺省值是 NULL 。这么做通常是合理的，因为 NULL 表示"未知"。

在一个表定义里，缺省值是在字段数据类型后面列出的。比如：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric **DEFAULT 9.99**
);
```

缺省值可以是一个表达式，它会在插入缺省值的时候计算(不是在创建表的时候)。 一个常见的例子是一个 `timestamp` 字段可能有缺省值 `CURRENT_TIMESTAMP` ， 它表示插入行的时刻。另外一个常见的例子是为每一行生成一个"序列号"。 在PostgreSQL里，通常是用类似下面这样的方法生成的：

```
CREATE TABLE products (
    product_no integer **DEFAULT nextval('products_product_no_seq')**,
    ...
);
```

这里的 `nextval()` 从一个序列对象(参阅Section 9.16) 提供后继的数值。这种做法非常普遍，以至于我们有一个专门的缩写用于此目的：

```
CREATE TABLE products (
    product_no **SERIAL**,
    ...
);
```

`SERIAL` 缩写在Section 8.1.4里有进一步描述。

5.3. 约束

数据类型是限制我们可以在表里存储什么数据的一种方法。不过，对于许多应用来说， 这种限制实在是太粗糙了。比如，一个包含产品价格的字段应该只接受正数。 但是没有哪种标准数据类型只接受正数。 另外一个问题是你可能需要根据其它字段或者其它行的数据来约束字段数据。比如， 在一个包含产品信息的表中，每个产品编号都应该只有一行。

对于这些问题，SQL 允许你在字段和表上定义约束。约束允许你对数据施加任意控制。 如果用户企图在字段里存储违反约束的数据，那么就会抛出一个错误。 这种情况同时也适用于数值来自缺省值的情况。

5.3.1. 检查约束

检查约束是最常见的约束类型。它允许你声明在某个字段里的数值必须使一个布尔表达式为真。 比如，要强制一个正数的产品价格，你可以用：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric **CHECK (price > 0)**
);
```

如你所见，约束定义在数据类型之后，就好像缺省值定义一样。缺省值和约束可以按任意顺序排列。 一个检查约束由一个关键字 `CHECK` 后面跟一个放在圆括弧里的表达式组成。 检查约束表达式应该包含受约束的字段，否则这个约束就没什么意义了。

你还可以给这个约束取一个独立的名字。这样就可以令错误信息更清晰， 并且在你需要修改它的时候引用这个名字。语法是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric **CONSTRAINT positive_price** CHECK (price > 0)
);
```

因此，要声明一个命名约束，使用关键字 `CONSTRAINT` 后面跟一个标识符(作为名字)， 然后再跟约束定义。如果你不用这个方法声明约束，那么系统会自动为你选择一个名字。

一个检查约束也可以引用多个字段。假设你存储一个正常价格和一个折扣价，并且你想保证折扣价比正常价低：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    **CHECK (price > discounted_price)**
);
```

头两个约束看上去很面熟。第三个使用了一个新的语法。它没有附着在某个字段上， 而是在逗号分隔的字段列表中以一个独立行的形式出现。字段定义和约束定义可以按照任意顺序列出。

我们称头两个约束是"字段约束"，而第三个约束是"表约束"(和字段定义分开写)。 字段约束也可以写成表约束，而反过来很可能不行，因为系统假设字段约束只引用它所从属的字段。PostgreSQL并不强制这条规则， 但是如果你希望自己的表定义可以和其它数据库系统兼容，那么你最好还是遵循这条规则。 上面的例子也可以这么写：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

或者是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

这只是风格的不同。

和字段约束一样，我们也可以给表约束赋予名称，方法也相同：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    **CONSTRAINT valid_discount** CHECK (price > discounted_price)
);
```

我们还要注意的是，当约束表达式计算结果为真或 NULL 的时候，检查约束会被认为是满足条件的。 因为大多数表达式在含有 NULL 操作数的时候结果都是 NULL ，所以这些约束不能阻止字段值为 NULL 。 要确保一个字段值不为 NULL ，可以使用下一节介绍的非空约束。

5.3.2. 非空约束

非空约束只是简单地声明一个字段必须不能是 NULL。下面是一个例子：

```
CREATE TABLE products (
    product_no integer **NOT NULL**,
    name text **NOT NULL**,
    price numeric
);
```

一个非空约束总是写成一个字段约束。非空约束在功能上等效于创建一个检查约束 `CHECK (``_column_name_` IS NOT NULL)， 但在PostgreSQL里，创建一个明确的非空约束效率更高。 缺点是你不能给它一个明确的名字。

当然，一个字段可以有多个约束。只要一个接着一个写就可以了：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

它们的顺序无所谓。顺序并不影响约束检查的顺序。

`NOT NULL` 约束有个相反的约束： `NULL` 约束。 它并不意味着该字段必须是空，因为这样的字段也没用。它只是定义了该字段可以为空的这个缺省行为。 在 SQL 标准里没有定义 `NULL` 约束，因此不应该在可移植的应用中使用它。 在PostgreSQL里面增加这个约束只是为了和其它数据库系统兼容。 不过，有些用户喜欢它，因为这个约束可以让他们很容易在脚本文件里切换约束。 比如，你可以从下面这样开始：

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

然后在需要的时候插入 `NOT` 关键字。

> **Tip:** 在大多数数据库设计里，主要的字段都应该标记为非空。

5.3.3. 唯一约束

唯一约束保证在一个字段或者一组字段里的数据与表中其它行的数据相比是唯一的。它的语法是：

```
CREATE TABLE products (
    product_no integer **UNIQUE**,
    name text,
    price numeric
);
```

上面是写成字段约束，下面这个则写成表约束：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    **UNIQUE (product_no)**
);
```

如果一个唯一约束引用一组字段，那么这些字段用逗号分隔列出：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    **UNIQUE (a, c)**
);
```

这样就声明了特定字段值的组合在整个表范围内是唯一的。 但是这些字段中的某个单独值可以不必是(并且通常也确实不是)唯一的。

你也可以给唯一约束赋予一个自己定义的名字，方法与前面相同：

```
CREATE TABLE products (
    product_no integer **CONSTRAINT must_be_different** UNIQUE,
    name text,
    price numeric
);
```

添加一个唯一约束通常会自动在约束中使用的列或一组列上创建一个唯一btree索引。

通常，如果包含在唯一约束中的那几个字段在表中有多个相同的行，就违反了唯一约束。 但是在这种比较中，NULL 被认为是不相等的。这就意味着，在多字段唯一约束的情况下， 如果在至少一个字段上出现 NULL， 那么我们还是可以存储同样的这种数据行。 这种行为遵循 SQL 标准，但是我们听说其它 SQL 数据库可能不遵循这个标准。 因此如果你要开发可移植的程序，那么最好仔细些。

5.3.4. 主键

从技术上讲，主键约束只是唯一约束和非空约束的组合。所以，下面两个表定义是等价的：

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer **PRIMARY KEY**,
    name text,
    price numeric
);
```

主键也可以约束多于一个字段；其语法类似于唯一约束：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    **PRIMARY KEY (a, c)**
);
```

主键表示一个或多个字段的组合可以用于唯一标识表中的数据行。 这是定义一个主键的直接结果。请注意：一个唯一约束实际上并不能提供一个唯一标识， 因为它不排除 NULL 。这个功能对文档目的和客户应用都很有用。比如， 一个可以修改行数值的 GUI 应用可能需要知道一个表的主键才能唯一地标识每一行。

添加一个主键将会在主键使用的列或一组列中自动创建一个唯一btree索引。

一个表最多可以有一个主键(但是它可以有多个唯一和非空约束)。关系型数据库理论告诉我们， 每个表都必须有一个主键。PostgreSQL并不强制这个规则， 但我们最好还是遵循它。

5.3.5. 外键

外键约束声明一个字段(或者一组字段)的数值必须匹配另外一个表中出现的数值。 我们把这个行为称为两个相关表之间的参照完整性。

假设你有个产品表，我们可能使用了好几次：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

假设你有一个存储这些产品的订单的表。我们想保证订单表只包含实际存在的产品。 因此我们在订单表中定义一个外键约束引用产品表：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer **REFERENCES products (product_no)**,
    quantity integer
);
```

现在，我们不能创建任何其非空 `product_no` 记录没有在产品表中出现的订单。

在这种情况下我们把订单表叫做引用表， 而产品表叫做被引用表。同样，也有引用字段和被引用字段。

你也可以把上面的命令简写成：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer **REFERENCES products**,
    quantity integer
);
```

因为如果缺少字段列表的话，就会引用被引用表的主键。

一个外键也可以约束和引用一组字段。同样，也需要写成表约束的形式。 下面是一个捏造出来的语法例子：

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  **FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)**
);
```

当然，被约束的字段数目和类型需要和被引用字段数目和类型一致。

和平常一样，你也可以给外键约束赋予自定义的名字。

一个表可以包含多于一个外键约束。这个特性用于实现表之间的多对多关系。 比如你有关于产品和订单的表，但现在你想允许一个订单可以包含多种产品 (上面那个结构是不允许这么做的)，你可以使用这样的结构：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

注意最后的表的主键和外键是重叠的。

我们知道外键不允许创建和任何产品都无关的订单。 但是如果一个订单创建之后其引用的产品被删除了怎么办？SQL 也允许你处理这个问题。 简单说，我们有几种选择：

- 不允许删除一个被引用的产品
- 同时也删除订单
- 其它的?

为了说明这个问题，我们对上面的多对多关系制定下面的策略： 如果有人想删除一种仍然被某个订单引用的产品(通过 `order_items`)， 那么就不允许这么做。如果有人删除了一个订单，那么订单项也被删除。

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products **ON DELETE RESTRICT**,
    order_id integer REFERENCES orders **ON DELETE CASCADE**,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

限制和级联删除是两种最常见的选项。 `RESTRICT` 禁止删除被引用的行。 `NO ACTION` 的意思是如果在检查约束的时候还存在任何引用行，则抛出错误； 如果你不声明任何东西，那么它就是缺省的行为。这两个选择的实际区别是： `NO ACTION` 允许约束检查推迟到事务的晚些时候，而 `RESTRICT` 不行。 `CASCADE` 声明在删除一个被引用的行的时候，所有引用它的行也会被

自动删除掉。在外键字段上的动作还有两个选项：`SET NULL` 和 `SET DEFAULT`，它们导致在被引用行删除的时候，将引用它们的字段分别设置为 NULL 和缺省值。请注意这些选项并不能让你逃脱被观察和约束的境地。比如，如果一个动作声明 `SET DEFAULT`，但是缺省值并不能满足外键约束，那么该动作就会失败。

与 `ON DELETE` 类似的还有 `ON UPDATE` 选项，它是在被引用字段修改(更新)的时候调用用的，可用的动作是一样的。在这种情况下，`CASCADE` 意味着被引用字段的更新后的值，应该被拷贝到引用行中。

通常地，如果一个引用行的任意引用字段为null，那么这个引用行不必满足外键约束。如果外键声明中添加了 `MATCH FULL`，引用行只有在所有的引用字段都是null时，才能逃避满足约束（所以null和non-null值的混合肯定不能满足 `MATCH FULL` 约束）。如果你不想引用行能够避免满足外键约束，那么声明引用行为 `NOT NULL`。

一个外键必须要么引用一个主键，要么引用一个唯一约束。这意味着被引用行总是有一个索引（一个基本的主键或唯一约束）；所以检查一个引用行是否有一个匹配是高效的。因此从被引用表中 `DELETE` 一个行或者一个被引用字段 `UPDATE` 一列，都需要扫描一次引用表以便从行中匹配老的数值，给引用字段创建索引也是一个好主意。因为这个不是总是被需要，而且怎么去建立索引还有许多其他的选择，外键约束的声明不能在引用字段上自动生成一个索引。

有关更新和删除数据的更多信息可以在Chapter 6里找到。也可以查看关于外键约束语法的描述，在参考文档的CREATE TABLE。

5.3.6. 排除约束

排他约束保证如果任何两行被在声明的字段里比较或者用声明的操作表达，至少有一个操作比较会返回错误或空值。句法是：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

更多细节也请参考 `CREATE TABLE ... CONSTRAINT ... EXCLUDE`。

添加一个排除约束会在约束声明里自动创建一个声明类型的索引。

5.4. 系统字段

每个表都有几个系统字段，这些字段是由系统隐含定义的。因此， 这些名字不能用于用户定义的字段名。请注意这些限制与这个名字是否关键字无关， 把名字用引号括起来并不能让你逃离这些限制。你实际上不需要注意这些字段； 只要知道它们存在就可以了。

`oid`

行对象标识符(对象ID)。这个字段只有在创建表的时候使用了 `WITH OIDS` 或者是配置参数default_with_oids的值为真时出现。 这个字段的类型是 `oid` (和字段同名)；参阅Section 8.18获取有关这种类型的更多信息。

`tableoid`

包含本行的表的OID。这个字段对那些从继承层次中选取的查询特别有用(参阅Section 5.8)，因为如果没有它的话，我们就很难说明一行来自哪个独立的表。 `tableoid` 可以和 `pg_class` 的 `oid` 字段连接起来获取表名字。

`xmin`

插入该行版本的事务标识(事务ID)。注意：在这个环境里， 一个行版本是一行的一个状态；一行的每次更新都为同一个逻辑行创建一个新的行版本。

`cmin`

在插入事务内部的命令标识(从零开始)。

`xmax`

删除事务的标识(事务ID)，如果不是被删除的行版本，那么是零。在一个可见行版本里， 这个字段有可能是非零。这通常意味着删除事务还没有提交，或者是一个删除的企图被回滚掉了。

`cmax`

删除事务内部的命令标识符，或者是零。

`ctid`

一个行版本在它所处的表内的物理位置。请注意，尽管 `ctid` 可以用于非常快速地定位行版本，但每次 `VACUUM FULL` 之后， 一个行的 `ctid` 都会被更新或者移动。因此 `ctid` 是不能作为长期的行标识符的。应该使用 OID ，或者更好是用户定义的序列号，来标识一个逻辑行。

OID是32位的量，是在同一个集群内通用的计数器上赋值的。 对于一个大型或者长时间使用的数据库，这个计数器是有可能重叠的。因此， 假定OID唯一是非常错误的，除非你自己采取了措施来保证它们是唯一的。 如果你需要标识表中的行，我们强烈建议使用序列号生成器。不过，也可以使用OID， 只要采取几个注意事项即可：

- 在使用OID标识行的每个表的OID字段创建一个唯一约束。在唯一约束(或者唯一索引)存在的时候， 系统会注意不去生成一个和现有行相同的OID。当然，只有在表中的数据行少于2³² (40亿)行的时候才是可能的，而实际上表中的行最好远比这个小，要不性能就会受影响了。
- 绝对不要假设OIDs是跨表唯一的；如果你需要全数据库范围内的标识， 请使用 `tableoid` 和行的OID的组合。
- 需要OID的表应该带着 `WITH OIDS` 创建。从PostgreSQL 8.1开始， `WITHOUT OIDS` 是缺省的。

事务标识符也是32位的量。在长时间运转的数据库里，它也可能会重叠。 只要我们采取一些合适的维护步骤，这并不是很要命的问题；参阅Chapter 23 获取细节。不过，在长时间运行的环境里(超过十亿次事务)依赖事务ID的唯一性并非明智的做法。

命令标识符也是32位的量。这样就在一个事务里有2³²(四十亿)条SQL命令的硬限制。 在现实里这个限制应该不是什么问题，需要注意的是这个限制是SQL命令的条数， 而不是处理的行版本的条数。而且，自PostgreSQL 8.3起， 只有真正修改数据库内容的命令才会消耗一个命令标识符。

5.5. 修改表

如果你创建了一个表后发现自己犯了一个错误，或者是应用的需求发生了变化， 那么你可以删除这个表然后重新创建它。但是如果这个表已经填充了许多数据， 或者该表已经被其它数据库对象引用(比如一个外键约束)，那这可不是一个方便的方法。 因此PostgreSQL提供了一族命令用于修改现有表。 请注意它在概念上和修改一个表中包含的数据是不一样的： 这里我们感兴趣的是修改一个表的定义，或者说结构。

你可以：

- 增加字段
- 删除字段
- 增加约束
- 删除约束
- 修改缺省值
- 修改字段数据类型
- 重命名字段
- 重命名表

所有这些动作都是用ALTER TABLE命令执行的， 它的参考页面包含超出这里给出的详细信息。

5.5.1. 增加字段

要增加一个字段，使用下面这样的命令：

```
ALTER TABLE products ADD COLUMN description text;
```

新增的字段对于表中已经存在的行而言最初将先填充所给出的缺省值 (如果你没有声明 `DEFAULT` 子句，那么缺省是NULL)。

你也可以同时在该字段上定义约束，使用通常的语法：

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

实际上，所有在 `CREATE TABLE` 里描述的可以应用于字段的选项都可以在这里使用。 不过，我们要注意的是缺省值必须满足给出的约束，否则 `ADD` 将会失败。另外， 你可以在正确填充了新字段的数值之后再增加约束(见下文)。

> **Tip:** 添加一个字段并填充缺省值将会导致更新表中的所有行(为了存储新字段的值)， 但如果没有声明缺省值，PostgreSQL就可以避免物理更新。 所以如果你将要在新字段中填充的值大多数都不等于缺省值， 那么最好添加一个没有缺省值的字段，然后再使用 `UPDATE` 更新数据， 最后使用下面的方法添加缺省值。

5.5.2. 删除字段

要删除一个字段，使用下面这样的命令：

```
ALTER TABLE products DROP COLUMN description;
```

不管字段里有啥数据，都会消失，和这个字段相关的约束也会被删除。不过， 如果这个字段被另一个表的外键约束所引用，PostgreSQL 则不会隐含地删除该约束。你可以通过使用 `CASCADE` 指明删除任何依赖该字段的东西：

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

参阅Section 5.12获取有关这些操作背后的机制的信息。

5.5.3. 增加约束

要增加一个约束，必须使用表约束语法。比如：

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

要增加一个不能写成表约束的非空约束，使用下面的语法：

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

这个约束将立即进行检查，所以表在添加约束之前必须符合约束条件。

5.5.4. 删除约束

要删除一个约束，你需要知道它的名字。如果你曾经给了它取了名字， 那么事情就很简单。否则你就需要找出系统自动分配的名字。psql 的命令 `\d` `_tablename_` 可以帮这个忙； 其它接口可能也提供了检查表的细节的方法。然后就是这条命令：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

如果你在处理一个生成的约束名，比如 `$2` ， 别忘了你需要给它添加双引号， 让它成为一个有效的标识符。

和删除字段一样，如果你想删除被依赖的约束，你需要用 `CASCADE` 。 一个例子是某个外键约束依赖被引用字段上的唯一约束或者主键约束。

除了非空约束外，所有约束类型都这么用。要删除非空约束，可以这样：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

要记得非空约束没有名字。

5.5.5. 改变字段的缺省值

要给一个字段设置缺省值，可以使用一个像下面这样的命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

请注意这么做不会影响任何表中现有的数据行，它只是为将来的 `INSERT` 命令改变缺省值。

要删除缺省值，可以用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这样实际上相当于把缺省设置为空。结果是， 如果我们删除一个还没有定义的缺省值不算错误，因为缺省隐含就是NULL。

5.5.6. 修改字段的数据类型

把一个字段转换成另外一种数据类型，使用下面的命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有在字段里现有的每个项都可以隐含的转换成新类型时才可能成功。 如果需要更复杂的转换，你可以增加一个 `USING` 子句，它声明如何从旧值里计算新值。

PostgreSQL将试图把字段的缺省值(如果存在)转换成新的类型， 还有涉及该字段的任何约束。但是这些转换可能失败，或者可能生成奇怪的结果。 在修改某字段类型之前，你最好删除那些约束，然后再把合适的约束添加上去。

5.5.7. 重命名字段

重命名一个字段：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. 重命名表

重命名一个表：

```
ALTER TABLE products RENAME TO items;
```

5.6. 权限

当创建一个数据库对象时，它就被赋予了所有者。这个所有者通常是执行创建语句的角色。对大多数类型的对象，初始状态只有其所有者（或者超级管理员）可以对它做任何事情。 要允许其他角色使用它，必须要经过权限授予。

有好多种不同的权限：`SELECT`，`INSERT`，`UPDATE`，`DELETE`，`TRUNCATE`，`REFERENCES`，`TRIGGER`，`CREATE`，`CONNECT`，`TEMPORARY`，`EXECUTE`，和 `USAGE`。适用于特定对象的权限因对象类型(表/函数等)不同而不同。 有关PostgreSQL所支持的不同类型的权限的完整信息，请参考GRANT的手册页。下面的章节将为你展示如何利用这些权限。

修改或者删除一个对象的权限永远是所有者独有的权限。

一个对象可以用 `ALTER` 命令以适当的对象类型赋予新的所有者， 例如ALTER TABLE。超级用户总是可以这么做；普通用户只有在他同时是当前对象的所有者（或者所有角色的一个成员） 和新所有者角色的成员时可以这样做。

使用 `GRANT` 命令赋予权限。例如，如果 `joe` 是一个已经存在的用户， 而 `accounts` 是一个已经存在的表，更新表的权限可以用下面的命令赋予：

```
GRANT UPDATE ON accounts TO joe;
```

在权限的位置写上 `ALL` 则赋予所有与该对象类型相关的权限。

名为 `PUBLIC` 的特殊"用户"可以用于将权限赋予系统中的所有用户。 另外，还可以使用"组"角色来帮助管理一群用户的权限，细节可参见Chapter 20。

可以使用 `REVOKE` 命令撤销权限：

```
REVOKE ALL ON accounts FROM PUBLIC;
```

对象所有者的特殊权限(也就是 `DROP`，`GRANT`，`REVOKE` 等权限)总是隐含地属于所有者，并且不能赋予或者撤销。 但是对象所有者可以选择撤销自己的普通权限，比如把一个表做成对自己和别人都是只读的。

最初，只有对象所有者(或者超级用户)可以赋予或者撤销对象的权限。但是， 我们可以赋予一个"with grant option"权限，这样就允许接受权限的人将该权限转授他人。 如果授权选项后来被撤销，那么所有那些从这个接受者接受了权限的用户(直接或间级)都将失去该权限。 细节详见GRANT和REVOKE手册页。

5.7. 模式

一个PostgreSQL数据库集群包含一个或多个已命名数据库。 用户和用户组在整个集群范围内是共享的，但是其它数据并不共享。 任何与服务器连接的客户都只能访问那个在连接请求里声明的数据库。

> **Note:** 集群中的用户并不一定要有访问集群内所有数据库的权限。共享用户名的意思是不能有重名用户。 假定同一个集群里有两个数据库和一个 `joe` 用户，系统可以配置成只允许 `joe` 访问其中的一个数据库。

一个数据库包含一个或多个已命名的模式，模式又包含表。模式还可以包含其它对象， 包括数据类型、函数、操作符等。同一个对象名可以在不同的模式里使用而不会导致冲突； 比如， `schema1` 和 `myschema` 都可以包含一个名为 `mytable` 的表。 和数据库不同，模式不是严格分离的：只要有权限，一个用户可以访问他所连接的数据库中的任意模式中的对象。

我们需要模式的原因有好多：

- 允许多个用户使用一个数据库而不会干扰其它用户。
- 把数据库对象组织成逻辑组，让它们更便于管理。
- 第三方的应用可以放在不同的模式中，这样它们就不会和其它对象的名字冲突。

模式类似于操作系统层次的目录，只不过模式不能嵌套。

5.7.1. 创建模式

要创建一个模式，使用CREATE SCHEMA命令。给出你选择的模式名字。比如：

```
CREATE SCHEMA myschema;
```

要创建或者访问在模式中的对象，写出一个受修饰的名字，这个名字包含模式名以及表名，它们之间用一个句点分开：

```
_schema_._table_
```

这个方式在任何需要表名字的地方都可用，包括后面章节讨论的表修改命令和数据访问命令。 出于简化，我们将只讨论表，这个概念适用于所有其它已命名对象类型，比如数据类型和函数。

实际上，更一般的语法

```
_database_._schema_._table_
```

也可以使用，但目前它只是为了和 SQL 标准形式上兼容。如果你写了一个数据库名， 那么它必须和你当前连接的数据库同名。

要在新模式里创建一个表，用

```
CREATE TABLE myschema.mytable (
 ...
);
```

如果一个模式是空的(所有它里面的对象都已经删除)，那么删除一个模式的命令如下：

```
DROP SCHEMA myschema;
```

要删除一个模式及其包含的所有对象，可以使用：

```
DROP SCHEMA myschema CASCADE;
```

参阅Section 5.12获取对隐藏在这些动作背后的东西的一般机制的描述。

通常你想创建一个他人拥有的模式(因为这是一种限制用户在定义良好的模式中的活动的方法)。其语法如下：

```
CREATE SCHEMA _schemaname_ AUTHORIZATION _username_;
```

你甚至可以省略模式名字，这时模式名将和用户名同名。参阅Section 5.7.6获取这种情况的适用场合。

以 `pg_` 开头的模式名是保留给系统使用的，用户不能创建这样的名字。

5.7.2. Public 模式

在前面的小节里，我们没有声明任何模式名字就创建了表。缺省时， 这样的表(以及其它对象)都自动放到一个叫做"public"的模式中去了。 每个新数据库都包含一个这样的模式。因此，下面的命令是等效的：

```
CREATE TABLE products ( ... );
```

和：

```
CREATE TABLE public.products ( ... );
```

5.7.3. 模式搜索路径

全称的名字写起来非常费劲，并且我们最好不要在应用里直接写上特定的模式名。因此， 表通常都是用未修饰的名字引用的，这样的名字里只有表名字。 系统通过查找一个搜索路径来判断一个表究竟是哪个表， 这个路径是一个需要查找的模式名列表。在搜索路径里找到的第一个表将被使用。 如果在搜索路径中没有找到表，那么就报告一个错误(即使在数据库里的其它模式中存在此表也如此)。

在搜索路径中的第一个模式叫做"当前模式"。除了是搜索的第一个模式之外， 它还是在 `CREATE TABLE` 没有声明模式名的时候，新建表的默认所在地。

要显示当前搜索路径，使用下面的命令：

```
SHOW search_path;
```

在缺省的设置中，返回下面的东西：

```
 search_path
--------------
 "$user",public
```

第一个元素声明搜索和当前用户同名的模式。因为还没有这样的模式存在， 所以这条记录被忽略。第二个元素指向我们已经看过的公共模式。

搜索路径中第一个存在的模式是创建新对象的缺省位置。这就是为什么缺省的对象都会创建在 public 模式里的原因。 如果在其它环境中引用对象且没有模式修饰，那么系统会遍历搜索路径，直到找到一个匹配的对象。 因此，在缺省的配置里，任何未修饰的访问只能引用 public 模式。

要设置模式的搜索路径，可以用(省略了 `$user` 是因为并不立即需要它)

```
SET search_path TO myschema,public;
```

然后我们就可以不使用模式修饰来访问表了：

```
DROP TABLE mytable;
```

同样，因为 `myschema` 是路径中的第一个元素，新对象缺省时将创建在这里。

我们也可以写成：

```
SET search_path TO myschema;
```

然后我们如果不明确修饰的话，就不能再访问 public 模式了。public 模式没有任何特殊之处， 只不过它缺省时就存在。我们也可以删除它。

又见Section 9.25以获取其它操作模式搜索路径的方法。

搜索路径对于数据类型名、函数名、操作符名的运作方式和表名完全相同。 数据类型和函数名可以像表名一样加以修饰。如果你需要在表达式里写一个有模式修饰的操作符， 你必须这么写：

```
OPERATOR(_schema_._operator_)
```

这样是为了避免语法歧义。下面是一个例子：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

实践中我们通常依赖搜索路径寻找操作符，这样就不用写这么难看的东西了。

5.7.4. 模式和权限

缺省时，用户无法访问模式中不属于他们所有的对象。为了让他们能够访问， 模式的所有者需要在模式上赋予他们 `USAGE` 权限。 为了让用户使用模式中的对象，我们可能需要赋予适合该对象的额外权限。

用户也可以在别人的模式里创建对象。要允许这么做，需要被赋予在该模式上的 `CREATE` 权限。 请注意，缺省时每个人都在 `public` 模式上有 `CREATE` 和 `USAGE` 权限。 这样就允许所有可以连接到指定数据库上的用户在这里创建对象。如果你不打算这么做，可以撤销这个权限：

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

第一个"public"是模式，第二个"public"意思是"所有用户"。 第一句里它是个标识符，而第二句里是个关键字，所以有不同的大小写。 记住我们在Section 4.1.1里面说过的原则。

5.7.5. 系统表模式

除了 `public` 和用户创建的模式之外，每个数据库都包含一个 `pg_catalog` 模式， 它包含系统表和所有内置数据类型、函数、操作符。 `pg_catalog` 总是搜索路径中的一部分。 如果它没有明确出现在路径中，那么它隐含地在所有路径之前搜索。 这样就保证了内置名字总是可以被搜索。不过，你可以明确地把 `pg_catalog` 放在搜索路径之后， 如果你想使用用户自定义的名字覆盖内置的名字的话。

在PostgreSQL版本 7.3 之前，以 `pg_` 开头的表名字是保留的。 这个规则现在不正确了：如果必要，你可以创建这样的表名字，只要是在非系统模式里。 不过，我们最好还是不要使用这样的名字，以保证自己将来不会和新版本冲突： 那些版本也许会定义一些和你的表同名的表(在缺省搜索路径中，一个对你的表的无修饰引用将解析为系统表)。 系统表将继续遵循以 `pg_` 开头的传统，因此，只要你的表不是以 `pg_` 开头， 就不会和无修饰的用户表名字冲突。

5.7.6. 使用方式

模式可以用多种方式组织数据。下面是一些建议使用的模式，它们也很容易在缺省配置中得到支持：

- 如果没有创建任何模式，那么所有用户隐含都访问 public 模式。这样就模拟了没有模式的时候的情景。 这种设置建议主要用在只有一个用户或者数据库里只有几个可信用户的情形。 这样的设置也允许我们平滑地从无模式的环境过渡。
- 你可以为每个用户创建一个模式，名字和用户相同。要记得缺省的搜索路径从 `$user` 开始， 它会解析为用户名。因此，如果每个用户都有一个独立的模式，那么他们缺省时访问他们自己的模式。

  如果你使用了这样的设置，那么你可能还想撤销对 public 模式的访问(或者删除它)， 这样，用户就真的限制于他们自己的模式了。
- 要安装共享的应用(被所有人使用的表、第三方提供的额外函数等等)， 我们可以把它们放到独立的模式中。只要记得给需要访问它们的用户赋予合适的权限就可以了。 然后用户就可以通过用一个模式名修饰来使用这些额外的对象，或者他们可以把额外的模式放到他们的搜索路径中。

5.7.7. 移植性

在 SQL 标准里，在同一个模式里的对象被不同的用户所有的概念是不存在的。而且， 有些实现不允许你创建和它们的所有者不同名的模式。实际上， 模式的概念和用户在那些只实现了标准中规定的基本模式支持的数据库系统里几乎是一样的。 因此，许多用户考虑对名字加以修饰，使它们真正由 `_username_` . `_tablename_` 组成。如果你为每个用户都创建了一个模式，这实际上就是PostgreSQL的行为。

同样，在 SQL 标准里也没有 `public` 模式的概念。为了最大限度地遵循标准， 你不应该使用(可能甚至是应该删除) `public` 模式。

当然，有些数据库系统可能根本没有模式，或者是通过允许跨数据库访问来提供模式的功能。 如果你需要在这些系统上干活，那么为了最大限度的移植性，应该根本不使用模式。

5.8. 继承

PostgreSQL实现了表继承， 这个特性对数据库设计人员来说是一个很有效的工具。SQL99及以后的标准定义了类型继承特性， 和我们在这里描述的很多特性有区别。

让我们从一个例子开始：假设我们试图制作一个城市数据模型。每个州都有许多城市， 但是只有一个首府。我们希望能够迅速检索任何州的首府。这个任务可以通过创建两个表来实现， 一个是州府表，一个是非州府表。不过，如果我们不管什么城市都想查该怎么办？继承的特性可以帮助我们解决这个问题。我们定义 `capitals` 表， 它继承自 `cities` 表：

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- 英尺
);
CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

在这种情况下， `capitals` 表继承它的父表 `cities` 中的所有属性。州首府有一个额外的 `state` 属性显示其所在的州。

在PostgreSQL里，一个表可以从零个或多个其它表中继承属性， 而且一个查询既可以引用一个表中的所有行， 也可以引用一个表及其所有后代表的行(后面这个是缺省行为)。比如， 下面的查询查找所有海拔 500 英尺以上的城市名，包括州首府：

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;
```

使用PostgreSQL教程里面的数据(参阅Section 2.1)，它返回：

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

另一方面，如果要找出不包括州首府的所有海拔超过 500 英尺的城市，查询应该是这样的：

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

`cities` 前面的 `ONLY` 表明该查询应该只针对 `cities` 而不包括其后代。许多我们已经讨论过的命令(`SELECT` , `UPDATE` 和 `DELETE`)都支持 `ONLY` 关键字。

你也可以在表名后面写一个 `*` 显示指定包括所有后代表：

```
SELECT name, altitude
    FROM cities*
    WHERE altitude > 500;
```

因为这个行为是默认的，所以写 `*` 并不是必须的（除非你已经改变了 sql_inheritance里面的配置选项）。然而，写 `*` 可以用于强调搜索额外的表。

有时候你可能想知道某个行版本来自哪个表。在每个表里我们都有一个 `tableoid` 系统属性可以告诉你源表是谁：

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

结果如下(你可能会得到不同的 OID)：

```
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

通过和 `pg_class` 做一个连接，就可以看到实际的表名字：

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

它返回：

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

对于 `INSERT` 或 `COPY` ，继承并不自动影响其后代表。 在我们的例子里，下面的 `INSERT` 语句将会失败：

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

我们可能希望数据被传递到 `capitals` 表里面去， 但这是不会发生的： `INSERT` 总是插入明确声明的那个表。 在某些情况下，我们可以使用规则进行重定向插入(参阅 Chapter 38)。 不过它不能对上面的例子有什么帮助，因为 `cities` 表并不包含 `state` 字段，因此命令在规则施加之前就会被拒绝掉。

所有父表的检查约束和非空约束都会自动被所有子表继承。 不过其它类型的约束(唯一、主键、外键约束)不会被继承。

一个子表可以从多个父表继承，这种情况下它将拥有所有父表字段的总和， 并且子表中定义的字段也会加入其中。如果同一个字段名出现在多个父表中， 或者同时出现在父表和子表的定义里，那么这些字段就会被"融合"， 这样在子表里就只有一个这样的字段。要想融合，字段的数据类型必须相同， 否则就会抛出一个错误。融合的字段将会拥有其父字段的所有检查约束， 并且如果某个父字段存在非空约束，那么融合后的字段也必须是非空的。

表继承通常使用带 `INHERITS` 子句的CREATE TABLE语句定义。 另外，一个已经用此方法定义的子表可以使用带 `INHERIT` 的ALTER TABLE 命令添加一个新父表。注意：该子表必须已经包含新父表的所有字段且类型一致， 此外新父表的每个约束的名字及其表达式都必须包含在此子表中。同样， 一个继承链可以使用带 `NO INHERIT` 的 `ALTER TABLE` 命令从子表上删除。 允许动态添加和删除继承链对基于继承关系的表分区(参见Section 5.9)很有用。

创建一个将要作为子表的新表的便利途径是使用带 `LIKE` 子句的 `CREATE TABLE` 命令。它将创建一个与源表字段相同的新表。 如果源表中存在约束，那么应该指定 `LIKE` 的 `INCLUDING CONSTRAINTS` 选项，因为子表必须包含源表中的 `CHECK` 约束。

任何存在子表的父表都不能被删除，同样，子表中任何从父表继承的字段或约束也不能被删除或修改。 如果你想删除一个表及其所有后代，最简单的办法是使用 `CASCADE` 选项删除父表。

ALTER TABLE会把所有数据定义和检查约束传播到后代里面去。另外， 只有在使用 `CASCADE` 选项的情况下，才能删除依赖于其他表的字段。 `ALTER TABLE` 在重复字段融合和拒绝方面和 `CREATE TABLE` 的规则相同。

请注意表访问权限是如何处理的。访问父表会自动访问在子表中的数据，而不需要更多的访问权限检查。 这保留了父表中数据的表现。然而，直接访问子表不会自动允许访问父表，要访问父表需要更进一步的权限被授予。

5.8.1. 警告

注意，不是所有的 SQL 命令可以在所有的继承层次上正常工作。数据查询，数据修改， 模式修改的命令（比如 `SELECT`， `UPDATE`， `DELETE`， `ALTER TABLE` 的大多数变型，但不是 `INSERT` 和 `ALTER TABLE ... RENAME` ）典型的默认包括子表和支持 `ONLY` 符号来排除它们。

为数据库维护和调优的命令（例如 `REINDEX` ， `VACUUM` ） 通常只对个别工作，物理表格不支持递归超过继承层次结构。单独命令各自的行为记录在了它们的参考页中(Reference I, *SQL 命令*)。

继承的一个严重局限性是索引(包括唯一约束)和外键约束只能用于单个表， 而不能包括它们的子表(不管对外键约束的引用表还是被引用表都是如此)，因此，在上面的例子里：

- 即使我们声明 `cities` . `name` 为 `UNIQUE` 或 `PRIMARY KEY` ， 也不会阻止 `capitals` 表拥有重复名字的 `cities` 数据行。 并且这些重复的行在查询 `cities` 表的时候会显示出来。实际上， 缺省时 `capitals` 将完全没有唯一约束，因此可能包含带有同名的多个行。 你应该给 `capitals` 增加唯一约束，但即使这样做也不能避免与 `cities` 的重复。

- 类似的，即使我们声明 `cities` . `name` 参照（ `REFERENCES` ） 某些其它的表，这个约束也不会自动传播到 `capitals` 表。在这种条件下， 你可以通过手工给 `capitals` 表增加同样的 `REFERENCES` 约束来做到这点。

- 声明一个其它表的字段为 `REFERENCES cities(name)` 将允许其它表包含城市名， 但是不包含首府名。这种情况下没有很好的绕开办法。

这些缺点很可能在将来的版本中修补，但同时你也需要考虑一下，继承是否对你的应用真正有用。

5.9. 分区

PostgreSQL支持基本的表分区功能。 本节描述为什么需要表分区以及如何在数据库设计中使用表分区。

5.9.1. 概述

分区的意思是把逻辑上的一个大表分割成物理上的几块。分区可以提供若干好处：

- 某些类型的查询性能可以得到极大提升。 特别是表中访问率较高的行位于一个单独分区或少数几个分区上的情况下。 分区可以减少索引体积从而可以将高使用率部分的索引存放在内存中。 如果索引不能全部放在内存中，那么在索引上的读和写都会产生更多的磁盘访问。
- 当查询或更新一个分区的大部分记录时， 连续扫描那个分区而不是使用索引离散的访问整个表可以获得巨大的性能提升。
- 如果需要大量加载或者删除的记录位于单独的分区上， 那么可以通过直接读取或删除那个分区以获得巨大的性能提升， 因为 `ALTER TABLE NO INHERIT` 和 `DROP TABLE` 比操作大量的数据要快的多。这些命令同时还可以避免由于大量 `DELETE` 导致的 `VACUUM` 超载。
- 很少用的数据可以移动到便宜一些的慢速存储介质上。

这种好处通常只有在表可能会变得非常大的情况下才有价值。 到底多大的表会从分区中收益取决于具体的应用， 不过有个基本的拇指规则就是表的大小超过了数据库服务器的物理内存大小。

目前，PostgreSQL支持通过表继承进行分区。 每个分区必须做为单独一个父表的子表进行创建。父表自身通常是空的， 它的存在只是为了代表整个数据集。你在试图实现分区之前，应该先熟悉继承(参阅Section 5.8)。

PostgreSQL可以实现下面形式的分区：

范围分区

表被一个或者多个关键字段分区成"范围"，这些范围在不同的分区里没有重叠。 比如，我们可以通过时间范围分区，或者根据特定业务对象的标识符范围分区。

列表分区

表通过明确地列出每个分区里应该出现哪些关键字值实现。

5.9.2. 实现分区

要设置一个分区的表，做下面的步骤：

1. 创建"主表"，所有分区都从它继承。

   这个表中没有数据，不要在这个表上定义任何检查约束，除非你希望约束同样也适用于所有分区。 同样，在其上定义任何索引或者唯一约束也没有意义。

2. 创建几个"子表"，每个都从主表上继承。通常，这些表不会增加任何字段。

   我们将把子表称作分区，尽管它们就是普通的PostgreSQL表。

3. 给分区表增加约束，定义每个分区允许的健值。

   典型的例子是：

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID &gt;= 100 AND outletID &lt; 200 )
   ```

   确保这些约束能够保证在不同的分区里不会有重叠的键值。一个常见的错误是设置下面这样的范围：

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

   这样做是错误的，因为它没说清楚健值 200 属于那个范围。

   请注意在范围和列表分区的语法方面没有什么区别；这些术语只是用于描述的。

4. 对于每个分区，在关键字字段上创建一个索引，以及其它你想创建的索引。 关键字字段索引并非严格必需的，但是在大多数情况下它是很有帮助的。 如果你希望关键字值是唯一的，那么你应该总是给每个分区创建一个唯一或者主键约束。

5. 另外，定义一个规则或者触发器，来重定向数据插入主表到适当的分区。

6. 确保 `postgresql.conf` 里的配置参数constraint_exclusion是打开的。 没有这个参数，查询不会按照需要进行优化。

比如，假设我们为一个巨大的冰激凌公司构造数据库。该公司每天都测量最高温度， 以及每个地区的冰激凌销售。概念上，我们需要一个这样的表：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

我们知道大多数查询都只会访问最后一周，最后一个月或者最后一个季度的数据， 因为这个表的主要用途是为管理准备在线报告。为了减少需要存储的旧数据， 我们决定值保留最近三年的有用数据。在每个月的开头，我们都会删除最旧的一个月的数据。

在这种情况下，我们可以使用分区来帮助实现所有对表的不同需求。 下面的步骤描述了上面的需求，分区可以这样设置：

1. 主表是 `measurement` 表，就像上面那样声明。
2. 然后我们为每个月创建一个分区：

```
CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
```

   每个分区都是拥有自己内容的完整的表，只是它们从 `measurement` 表继承定义。

   这样就解决了我们的一个问题：删除旧数据。每个月， 我们需要做的只是在最旧的子表上执行一个 `DROP TABLE` ， 然后为新月份创建一个新的子表。

3. 我们必须提供非重叠的表约束。而不是只像上面那样创建分区表，所以我们的建表脚本就变成：

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate &gt;= DATE '2006-02-01' AND logdate &lt; DATE '2006-03-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate &gt;= DATE '2006-03-01' AND logdate &lt; DATE '2006-04-01' )
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate &gt;= DATE '2007-11-01' AND logdate &lt; DATE '2007-12-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate &gt;= DATE '2007-12-01' AND logdate &lt; DATE '2008-01-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate &gt;= DATE '2008-01-01' AND logdate &lt; DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 我们可能还需要在关键字字段上有索引：

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

   我们选择先不建立更多的索引。

5. 我们想让我们的应用可以说 `INSERT INTO measurement ...` 并且数据被重定向到相应的分区表。我们可以安排给主表附上一个合适的触发器。 如果数据只进入最新的分区，我们可以使用一个非常简单的触发器：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

创建完函数后，我们将创建一个调用触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

我们必须每月重新定义触发器，以便它总是指向当前分区。然而，触发定义不需要更新。

我们可能想插入数据并且想让服务器自动定位应该向哪个分区插入数据。 我们可以用下面这个复杂的触发器来实现这个目标，比如：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate &gt;= DATE '2006-02-01' AND
         NEW.logdate &lt; DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate &gt;= DATE '2006-03-01' AND
            NEW.logdate &lt; DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate &gt;= DATE '2008-01-01' AND
            NEW.logdate &lt; DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement_insert_trigger() fur
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

每一个触发器跟以前一样。注意，每一个 `IF` 测试必须匹配其分区的 `CHECK` 约束。

当这个函数比单月的情况更复杂时，它不需要经常的更新，因为分支可以在需要之前被添加。

> **Note:** 在实践中，如果大部分插入该分区，它可能最好首先检查最新分区。 为简单起见，我们已经在这个例子中的其他部分表明在同一顺序下的触发器的测试。

我们可以看出，一个复杂的分区方案可能要求相当多的 DDL 。 在上面的例子里我们需要每个月创建一次新分区， 因此写一个脚本自动生成需要的 DDL 是明智的。

5.9.3. 管理分区

通常分区集在定义表的时候就已经确定了，但我们常常需要周期性的删除旧分区并添加新分区。 分区最重要的好处是它能恰到好处的适应这个需求：以极快的速度操作分区的结构， 而不是痛苦的物理移动大量数据。

删除旧数据最简单的方法是删除不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

这个命令可以迅速删除数包含数百万条记录的分区，因为它不需要单独删除每一条记录。

还可以在删除分区的同时保留其作为一个表访问的能力：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

这将允许将来对这些数据执行其它的操作(比如使用 `COPY` ,pg_dump 之类的工具进行备份)。并且此时也是执行其它数据操作(数据聚集或运行报表等)的有利时机。

同样，我们可以像前面创建最初的分区一样，创建一个新的空分区来处理新数据。

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

有时在分区结构之外创建新表并在一段时间之后将其变为分区更为方便。 因为这将允许在该表变为分区之前对其中的数据进行加载、检查、转换之类的操作。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
--其它可能的数据准备工作
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.9.4. 分区和约束排除

约束排除是一种查询优化技巧，它改进了用上述方法定义的表分区的性能。比如：

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

如果没有约束排除，上面的查询会扫描 `measurement` 表中的每一个分区。 打开了约束排除之后，规划器将检查每个分区的约束然后试图证明该分区不需要被扫描 (因为它不能包含任何符合 `WHERE` 子句条件的数据行)。如果规划器可以证明这个， 它就把该分区从查询规划里排除出去。

你可以使用 `EXPLAIN` 命令显示一个规划在 `constraint_exclusion` 打开和关闭情况下的不同。一个为这种类型的表设置的典型的非最佳的规划是：

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

                                          QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m02 measurement  (cost=0.00..30.38 rows=543 wid
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03 measurement  (cost=0.00..30.38 rows=543 wid
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m12 measurement  (cost=0.00..30.38 rows=543 wid
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 wid
               Filter: (logdate >= '2008-01-01'::date)
```

部分或者全部分区可能会使用索引扫描而不是全表扫描， 不过这里要表达的意思是没有必要扫描旧分区就可以回答这个查询。 在打开约束排除之后，我们可以得到生成同样回答的明显简化的规划：

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                          QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 wid
               Filter: (logdate >= '2008-01-01'::date)
```

请注意，约束排除只由 `CHECK` 约束驱动，而不会由索引驱动。 因此，在关键字字段上定义索引是没有必要的。 在给出的分区上是否需要建立索引取决于那些扫描该分区的查询通常是扫描该分区的一大部分还是只是一小部分。 对于后者，索引通常都有帮助，对于前者则没有什么好处。

constraint_exclusion缺省(和建议）设置事实上不是 `on` 也不是 `off`， 但是中间设置调用 `partition`， 导致很可能要工作在分区表上的技术只适用于查询。 `on` 设置导致规划器在所有的查询里检查 `CHECK` 限制，即使是不可能受益的最简单的限制。

5.9.5. 替代分区方法

用一个不同的途径去重新定向插入适当的分区表是在主表中建立规则，而不是触发器，例如：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

规则比触发器有显著的开销，但是这个开销是每检查一次支付一次而不是每行支付一次， 所以这种方法可能在批量插入的情况下有优势。然而在更多的情况下， 触发器的方法更好。

请注意 `COPY` 会忽略规则。如果您想用 `COPY` 插入数据， 您将需要复制分区表而不是主表。 `COPY` 触发触发器， 如果您用触发器的方法就可以正常使用。

另一个规则方法缺点是如果规则设置没有覆盖插入数据，那么没有简单的路径强制错误， 数据将会悄悄代替主表中的数据。

安排分区也可以用 `UNION ALL` 视图，而不是表继承。例如，

```
CREATE VIEW measurement AS
          SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

然而，增加和删除各个分区的数据集，需要重新创建视图，增加一个额外的步骤。 在实际中这个方法跟使用继承相比较几乎没有可取之处。

5.9.6. 警告

下面的注意事项适合于已分区的表：

- 没有办法自动验证所有的 `CHECK` 约束是互斥的。 创建代码比每条用手生成分区和创建和/或修改关联的对象写更安全。
- 这里显示的模式假设分区内一行的主字段永远不变，或者至少不变足够要求它移到另一个分区。 一个 `UPDATE` 尝试由于 `CHECK` 的约束将会失败。 如果您需要处理这种情况，您可以在分区表内放入合适的更新触发器， 但是它会使管理结构更加复杂。

- 如果您正在使用`VACUUM`手册或者`ANALYZE`命令， 不要忘记您需要在每个分区上分别运行他们，就像这样的命令：

  ```
  ANALYZE measurement;
  ```

  将只会处理主表。

下面的注意事项适合于约束排除：

- 约束排除只是在查询的`WHERE`子句包含常量（或者外部提供的参数）的时候才生效。 例如，一个非不可变的函数的比较，如`CURRENT_TIMESTAMP`不能被优化，因为在运行时规划器不知道该参数会选择哪个分区。
- 保持分区约束的简单性，否则规划器可能不能证明分区不需要被访问。 为列表分区使用简单平等的约束，或为范围分区使用简单的范围测试，就像前面的例子说明。 一个好的拇指规则是分区约束应该只包含分区字段和可添加B-tree索引的操作符使用的常量的比较。
- 主表的所有分区的所有约束在约束排除中被审查，所以大量的分区将大大增加查询规划时间。 分区使用这些技术或许可以将分区提升到一百个且能很好的工作； 不要试图使用成千上万的分区。

5.10. 外部数据

PostgreSQL实现了SQL/MED规范的一部分，允许使用规则的SQL查询访问驻留在PostgreSQL外部的数据。这样的数据被称为外部数据。（请注意这种使用不能同外键混淆，外键是数据库的一种约束类型。）

外部数据是通过外部数据封装器的帮助来访问的。一个外部数据封装器是一个可以与外部数据源沟通的库，隐藏与外部数据源连接的细节并且从外部数据源获得数据。这里有几个作为`贡献`模板的可用外部数据封装器，见Appendix F。其他类型的外部数据封装器可能会在第三方产品中见到。如果现存的外部数据封装器没有适合你的需要的，你可以自己写一个；见Chapter 52。

要访问外部数据，你需要创建一个外部服务器对象，它定义了如何根据支持的外部数据封装器设置的选项，连接到一个特定的外部数据源。然后你需要创建一个或多个外部表，它定义了远程数据的结构。一个外部表可以像普通表那样用于查询，但是外部表不会存储在PostgreSQL服务器中。无论何时用到外部表，PostgreSQL要求外部数据封装器从外部源获取数据，或者在更新命令时传输数据到外部源。

访问远程数据可能需要到外部数据源的验证。这个信息可以通过一个用户映射来提供，用户映射可以根据当前的PostgreSQL角色提供额外的数据，如用户名和密码。

要获取更多的信息，请参阅 CREATE FOREIGN DATA WRAPPER，CREATE SERVER，CREATE USER MAPPING和 CREATE FOREIGN TABLE。

5.11. 其它数据库对象

在关系结构里，表是核心的对象，因为它们保存你的数据。但是它们并非存在于数据库中的唯一对象。 我们可以创建许多其它类型的对象来让我们对数据的使用和管理变得更方便。 我们没有在这一章里讨论这些对象，但是我们在这里会给你一个列表，这样你就知道什么是可能的。

- 视图
- 函数和操作符
- 数据类型和域
- 触发器和重写规则

这些主题的详细信息在Part V里面。

5.12. 依赖性跟踪

如果你创建了一个包含许多表，并且带有外键约束、视图、触发器、函数等复杂的数据库结构。 那么你就会在对象之间隐含地创建了一个依赖性的网络。比如，一个带有外键约束的表依赖于它所引用的表。

为了保证整个数据库结构的完整性，PostgreSQL 保证你无法删除那些还被其它对象依赖的对象。比如，试图删除在Section 5.3.5 里被订单表所依赖的产品表是不能成功的，会有类似下面的错误信息出现：

```
DROP TABLE products;

NOTICE:  constraint orders_product_no_fkey on table orders depends on table products
ERROR:  cannot drop table products because other objects depend on it
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

这个错误信息包含一个有用的提示：如果你不想麻烦的分别删除所有依赖对象，你可以运行：

```
DROP TABLE products CASCADE;
```

然后所有被依赖的对象都将被删除(并不删除订单表，只是删除外键约束)。 如果你想检查 `DROP ... CASCADE` 会干什么，运行不带 `CASCADE` 的 `DROP` 然后阅读 `NOTICE` 信息。

PostgreSQL里的所有删除命令都支持声明 `CASCADE` 。 当然，具体的依赖性实体取决于对象的类型。你也可以写 `RESTRICT` 而不是 `CASCADE` 以获取缺省的行为(防止删除那些其它对象所依赖的对象)。

> **Note:** 根据 SQL 标准，要求至少声明 `RESTRICT` 或 `CASCADE` 中的一个。 实际上没有哪种数据库系统强制这一点，但是缺省的行为是 `RESTRICT` 还是 `CASCADE` 则因系统而异。
>
> **Note:** 在PostgreSQL 7.3之前的外键约束依赖性和序列字段依赖性在升级过程中都不会得到维护或者创建。 所有其它的依赖性类型在从7.3版本以前的数据库升级过程中都将得到恰当的创建。

Chapter 6. 数据操作

Table of Contents

前面的章节讨论了如何创建存储数据的表和其它结构。现在来讲用数据填充表。 本章将介绍如何插入、更新、删除数据。 在下一章将最终解释如何把你丢失已久的数据从数据库中抽取出来。

6.1. 插入数据

在创建完一个表的时候，它里面没有数据。 在数据库可以有点用处之前要做的第一件事就是向里面插入数据。 数据在概念上是每次插入一行。我们当然可以每次插入多行， 但是确实没有办法插入少于一行的数据。即使你只知道几个字段的数值， 那么你也必须创建一个完整的行。

使用INSERT命令创建一个新行。 这条命令要求提供表名字以及字段值。比如， 考虑来自Chapter 5的产品表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

下面是一个向表中插入一行的例子：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据值是按照这些字段在表中出现的顺序列出的，并且用逗号分隔。通常， 数据值是文本(常量)，但也允许使用标量表达式。

上述语法的缺点是你必须知道表中字段的顺序。 你也可以明确地列出字段以避免这个问题。比如， 下面的两条命令都和上面的那条命令效果相同：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为明确列出字段名是个好习惯。

如果你不知道所有字段的数值，那么可以省略其中的一些。这时候， 这些未知字段将被填充为它们的缺省值。比如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是PostgreSQL的一个扩展。 它从左向右用给定的值尽可能多的填充字段，剩余的填充缺省值。

为了保持清晰，你也可以对独立的字段或者整个行明确使用缺省值：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

你可以在一条命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

> **Tip:** 要一次插入大量数据，可以看看COPY命令。 它不像INSERT命令那么灵活，但是更高效。 参考Section 14.4获取更多有关装载海量数据的信息。

6.2. 更新数据

修改已经存储在数据库中的数据的行为叫做更新。你可以更新单独的一行， 也可以更新表中所有的行，还可以更新其中的一部分行。我们可以独立地更新每个字段， 而其它的字段则不受影响。

要更新现有的行，使用UPDATE命令。这需要三种信息：

1. 表的名字和要更新的字段名
2. 字段的新值
3. 要更新哪些行

我们在Chapter 5里说过，SQL 通常并不为数据行提供唯一标识。 因此我们无法直接声明需要更新哪一行。但是，我们可以通过声明一个被更新的行必须满足的条件。 只有在表里存在主键的时候 (不依赖于你叫它什么)，我们才能通过选取主键可靠地指定一个独立的行。 图形化的数据库访问工具依赖这个东西来让我们可以独立地更新某些行。

比如，这条命令将所有价格为 5 的产品重定价为 10 ：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这样做可能导致零行、一行或多行数据被更新。 如果我们试图执行一个不匹配任何行的更新，那也不算错。

让我们仔细看看这个命令。首先是关键字 `UPDATE` 跟着表名字。 和平常一样，表名字也可以是用模式修饰的，否则就会从模式路径中把它找出来。 然后是关键字 `SET` 跟着字段名与一个等号以及新的字段值。 新的字段值可以是任意标量表达式，而不仅仅是常量。比如， 如果你想把所有产品的价格提高 10% ， 可以用：

```
UPDATE products SET price = price * 1.10;
```

如你所见，新值的表达式也可以引用行中现有的数值。我们还忽略了 `WHERE` 子句。如果我们忽略了这个子句，那么表中所有的行都要被更新。 如果出现了 `WHERE` 子句，那么只有匹配其条件的行才会被更新。 请注意在 `SET` 子句中的等号是一个赋值，而在 `WHERE` 子句中的等号是比较，不过这样并不会导致任何歧义。当然 `WHERE` 条件不一定非得是相等测试。许多其它操作符也都可以使用(参阅Chapter 9)。 但是表达式必须得出一个布尔结果。

你还可以在一个 `UPDATE` 命令中更新更多的字段， 方法是在 `SET` 子句中列出更多赋值。比如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. 删除数据

到目前为止我们已经解释了如何向表中增加数据以及如何改变数据。 剩下的是讨论如何删除不再需要的数据。和前面增加数据一样， 删除数据也必须是从表中整行整行地删除。 在上一节里我们提到了 SQL 不提供直接访问独立行的方法。因此， 删除行只能是通过声明被删除行必须匹配的条件进行。如果你在表上有一个主键， 那么你可以声明准确的行。当然，你也可以删除匹配条件的一组行， 或者一次删除表中的所有行。

我们使用DELETE命令删除行。它的语法和 `UPDATE` 命令非常类似。比如，要从产品表中删除所有价格为 10 的产品，用：

```
DELETE FROM products WHERE price = 10;
```

如果你只是写：

```
DELETE FROM products;
```

那么表中所有行都会被删除！程序员一定要注意。

Chapter 7. 查询

Table of Contents

前面的章节解释了如何创建表，如何用数据填充它们，以及如何操作那些数据。 现在我们终于可以讨论如何从数据库中检索数据了。

7.1. 概述

从数据库中检索数据的过程或命令叫做查询。 在 SQL 里SELECT命令用于声明查询。 `SELECT` 命令的通用语法如下：

```
[WITH `_with_queries_`] SELECT _select_list_ FROM _table_expression_ [`_sort_specificatio
```

随后的几节将描述选择列表、表表达式、排序声明的细节。 `WITH` 查询被视为最后的，因为它们是一种先进的功能。

简单的查询的形式如下：

```
SELECT * FROM table1;
```

假设有一个 `table1` 表，这条命令将从 `table1` 中检索所有行和所有用户定义的字段。检索的方法取决于客户端应用程序。 比如，psql程序将在屏幕上显示一个 ASCII 艺术构成的表格，客户端库将提供检索独立行和字段的函数。选择列表声明为 `*` 表示表表达式提供的所有可用字段。 一个选择列表也可以选择可用字段的一个子集或者使用这些字段进行计算； 比如，如果 `table1` 有名为 `a` , `b` , `c` 的字段(可能还有其它)，那么你可以用下面的查询(假设 b 和 c 都是数字数据类型)：

```
SELECT a, b + c FROM table1;
```

参阅Section 7.3获取更多细节。

`FROM table1` 是一种简单的表表达式：它只读取了一个表。 通常，表表达式可以是基本表、连接、子查询的复杂构造。 但你也可以省略表表达式而只用 `SELECT` 命令当做一个计算器：

```
SELECT 3 * 4;
```

如果选择列表里的表达式返回变化的结果，那么这个东西就更有用了。比如， 你可以用这个方法调用函数：

```
SELECT random();
```

7.2. 表表达式

表表达式计算一个表，它包含一个 `FROM` 子句， 该子句可以根据需要选用 `WHERE` , `GROUP BY` , `HAVING` 子句。大部分表表达式只是指向磁盘上的一个所谓的基本表， 但是我们可以用更复杂的表达式以各种方法修改或组合基本表。

表表达式里的 `WHERE` , `GROUP BY` , `HAVING` 子句声明一系列对源自 `FROM` 子句的表的转换操作。 所有这些转换最后生成一个虚拟表，传递给选择列表计算输出行。

7.2.1. `FROM` 子句

FROM 子句 从一个逗号分隔的表引用列表中生成一个虚拟表。

```
FROM _table_reference_ [, `_table_reference_` [, ...]]
```

表引用可以是一个表名字(可能有模式修饰)或者是一个生成的表， 比如子查询、表连接、或这些东西的复杂组合。如果在 `FROM` 子句中列出了多于一个表， 那么它们交叉连接(见下文)形成一个派生表，该表可以进行 `WHERE` , `GROUP BY` , `HAVING` 子句的转换处理，并最后生成表表达式的结果。

如果一个表引用是一个简单的父表的名字，那么将包括其所有后代子表的行， 除非你在该表名字前面加 `ONLY` 关键字(这样任何子表都会被忽略)。

除了在表名字前面加 `ONLY` ，你可以在表名字后面写 `*` 明确指定包括后代表。写 `*` 不是必须的，因为这个行为是默认的 （除非你已经改变了sql_inheritance配置选项里面的设置）。 然而写 `*` 可能对于强调搜索额外的表是有用的。

7.2.1.1. 连接表

一个连接表是根据特定的连接规则从两个其它表(真实表或生成表)中派生的表。 我们支持内连接、外连接、交叉连接。

连接类型

交叉连接

```
_T1_ CROSS JOIN _T2_
```

对每个来自 `_T1_` 和 `_T2_` 的行进行组合（也就是，一个笛卡尔积），连接成的表将包含这样的行： 所有 `_T1_` 里面的字段后面跟着所有 `_T2_` 里面的字段。如果两表分别有 N 和 M 行，连接成的表将有 N*M 行。

`FROM` `_T1_` CROSS JOIN `_T2_` 等效于`FROM` `_T1_`,`_T2_`。 它还等效于`FROM` `_T1_` INNER JOIN `_T2_` ON TRUE(见下文)。

条件连接

```
_T1_ { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN _T2_ ON _boolean_expression_
_T1_ { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN _T2_ USING ( _join column list_ )
_T1_ NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN _T2_
```

`INNER` 和 `OUTER` 对所有连接类型都是可选的。 `INNER` 为缺省。 `LEFT` , `RIGHT` , 和 `FULL` 隐含外连接。

连接条件在 `ON` 或 `USING` 子句里声明， 或者用关键字 `NATURAL` 隐含地声明。 连接条件判断来自两个源表中的那些行是"匹配"的，这些我们将在下面详细解释。

`ON` 子句是最常见的连接条件的类型：它接收一个和 `WHERE` 子句相同的布尔表达式。如果两个分别来自 `_T1_` 和 `_T2_` 的行在 `ON` 表达式上运算的结果为真，那么它们就算是匹配的行。

`USING` 是一个连接条件的缩写语法：它接收一个用逗号分隔的字段名列表， 这些字段必须是连接表共有的并且其值必须相同。最后， `JOIN USING` 会将每一对相等的输入字段输出为一个字段，其后跟着所有其它字段。 因此， `USING (a, b, c)` 等效于 `ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)` 只不过是如果使用了 `ON` ， 那么在结果里 `a` ， `b` 和 `c` 字段都会有两个，而用 `USING` 的时候就只会有一个（如果使用了 `SELECT *` 的话，他们会优先发生）。

最后， `NATURAL` 是 `USING` 的缩写形式： 它自动形成一个由两个表中同名的字段组成的 `USING` 列表(同名字段只出现一次)。 如果没有同名的字段， `NATURAL` 的行为会像 `CROSS JOIN` 。

条件连接可能的类型是：

`INNER JOIN`

内连接。对于 T1 中的每一行 R1 ， 如果能在 T2 中找到一个或多个满足连接条件的行， 那么这些满足条件的每一行都在连接表中生成一行。

`LEFT OUTER JOIN`

左外连接。首先执行一次内连接。然后为每一个 T1 中无法在 T2 中找到匹配的行生成一行，该行中对应 T2 的列用 NULL 补齐。因此，生成的连接表里总是包含来自 T1 里的每一行至少一个副本。

`RIGHT OUTER JOIN`

右外连接。首先执行一次内连接。然后为每一个 T2 中无法在 T1 中找到匹配的行生成一行，该行中对应 T1 的列用 NULL 补齐。因此，生成的连接表里总是包含来自 T2 里的每一行至少一个副本。

`FULL OUTER JOIN`

全连接。首先执行一次内连接。然后为每一个 T1 与 T2 中找不到匹配的行生成一行， 该行中无法匹配的列用 NULL 补齐。因此，生成的连接表里无条件地包含 T1 和 T2 里的每一行至少一个副本。

如果 `_T1_` 和 `_T2_` 之一或全部是可以连接的表，那么所有类型的连接都可以串连或嵌套在一起。 你可以在 `JOIN` 子句周围使用圆括弧来控制连接顺序，如果没有圆括弧， 那么 `JOIN` 子句从左向右嵌套。

为了解释这些问题，假设我们有一个表 `t1`：

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

和 `t2`：

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

然后我们用不同的连接方式可以获得各种结果：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 CROSS JOIN t2;</kbd
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 INNER JOIN t2 ON t1
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 INNER JOIN t2 USING
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 NATURAL INNER JOIN
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 LEFT JOIN t2 ON t1.
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 LEFT JOIN t2 USING
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 RIGHT JOIN t2 ON t1
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 FULL JOIN t2 ON t1.
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

用 `ON` 声明的连接条件也可以包含与连接不直接相关的条件。 这种功能可能对某些查询很有用，但是需要我们仔细想清楚。比如：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 LEFT JOIN t2 ON t1.
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

请注意，将限制放在在 `WHERE` 子句中将会产生不同的结果：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM t1 LEFT JOIN t2 ON t1.
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

这是因为限制放在 `ON` 子句中时是先于连接处理的， 而限制放在 `WHERE` 子句中时是后于连接处理的。

7.2.1.2. 表和列别名

你可以给表或复杂的表引用起一个临时的表别名，以便被其余的查询引用。

要创建一个表别名，可以这样：

```
FROM _table_reference_ AS _alias_
```

或：

```
FROM _table_reference_ _alias_
```

`AS` 关键字没啥特别的含义。 `_alias_` 可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识，好让连接子句更易读一些。比如：

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num
```

取了别名之后就不允许再用最初的名字了。因此，这是不合法的：

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- wrong
```

表别名主要是为了方便标记，但对于自连接却是必须的。比如：

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

另外，要引用子查询的结果也必须使用别名(参见Section 7.2.1.3)。

圆括弧用于解决歧义。下面的第一个语句把别名 `b` 赋予第二个 `my_table` 表；而第二个语句则把别名 `b` 赋予了连接的结果。

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种形式的表别名除了给表赋予别名外，还给该表的字段也赋予了别名：

```
FROM _table_reference_ [AS] _alias_ ( _column1_ [, `_column2_` [, ...]] )
```

如果声明的字段别名比表里实际的字段少，那么后面的字段就没有别名。 这个语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个 `JOIN` 子句的输出结果附加了一个别名， 那么该别名就在 `JOIN` 里隐藏了其原始的名字。比如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL， 但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：别名 `a` 在别名 `c` 的外面是看不到的。

7.2.1.3. 子查询

子查询的结果(派生表)必须包围在圆括弧里并且必须赋予一个别名(参阅 Section 7.2.1.2)。比如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个例子等效于 `FROM table1 AS alias_name` 。 更有趣的例子是在子查询里面有分组或聚集的时候， 这个时候子查询不能归纳成一个简单的连接。

子查询也可以是一个 `VALUES` 列表：

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

这种情况同样也必须要取一个别名。还可以为 `VALUES` 列表中的字段取别名， 并且被认为是一个好习惯。更多信息参见Section 7.7。

7.2.1.4. 表函数

表函数是那些生成一个行集合的函数，这个集合可以是由基本数据类型(标量类型)组成， 也可以是由复合数据类型(表的行)组成。他们的用法类似一个表、视图、或 `FROM` 子句里的子查询。表函数返回的字段可以像一个表、视图、或者子查询字段那样包含在 `SELECT` , `JOIN` , `WHERE` 子句里。

如果表函数返回基本数据类型，那么单一结果字段的名字匹配函数名。 如果表函数返回复合数据类型，那么多个结果字段的名字和该类型的每个属性的名字相同。

可以在 `FROM` 子句中为表函数取一个别名，也可以不取别名。 如果一个函数在 `FROM` 子句中没有别名，那么将使用函数名作为结果表的名字。

一些例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

有时侯，把一个函数定义成根据不同的调用方法可以返回不同的字段是很有用的。 为了支持这个，表函数可以声明为返回伪类型 `record` 。如果在查询里使用这样的函数， 那么我们必须在查询中声明预期的行结构，这样系统才知道如何分析和规划该查询。 让我们看看下面的例子：

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

dblink函数（dblink模块的一部分）执行一个远程的查询。它声明为返回 `record`，因为它可能会被用于任何类型的查询。实际的字段集必须在调用它的查询中声明，这样分析器才知道类似 `*` 这样的东西应该扩展成什么样子。

7.2.1.5. `LATERAL` 子查询

`FROM` 子句中出现的子查询可以放在关键字 `LATERAL` 之前。这样就允许它们引用通过前置 `FROM` 条目提供的字段。（如果没有 `LATERAL`， 那么每个子查询都被认为是独立的并且不能交叉引用任何其他的 `FROM` 条目。）

`FROM` 中出现的表函数也可以出现在关键字 `LATERAL` 之前， 但是对于函数来说，这个关键字是可选的；函数的参数在任何情况下都可以包含通过前置 `FROM` 条目提供的字段。

`LATERAL` 条目可以出现在 `FROM` 列表的顶级，或者在 `JOIN` 树中。在后者的情况下，它在 `JOIN` 右侧时也可以参考左侧的条目。

当 `FROM` 包含 `LATERAL` 交叉引用时，评估收益如下： `FROM` 条目的每行或多个 `FROM` 条目的行组提供交叉引用的字段， `LATERAL` 条目被评估为使用行或行组的字段值。结果行像平常一样加入他们的计算行。这些来自字段原表中的行或行组就这样重复。

一个 `LATERAL` 常见的例子是：

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

这并不是特别有用的，因为它的结果正好和更传统做法的相同。

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

`LATERAL` 当交叉引用字段对于加入的计算行是重要的时是主要有用的。一个常见的应用是为一个set-returning函数提供一个参数值。例如，假设 `vertices(polygon)` 返回一个多边形的顶点坐标，我们可以识别出多边形的顶点距离近的存储在一个表中：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

这条语句也可以写成：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

或者几个其他等价的形式。（就像之前提到的，`LATERAL` 关键字在这个例子中不是必须的，但是我们为了明确而是用它。）

`LEFT JOIN` 对于 `LATERAL` 子查询来说往往是特别有用的， 所以即使 `LATERAL` 子查询不产生行，源行也将出现在结果中。例如， 如果 `get_product_names()` 返回一个制造商制造的产品名字， 但是一些在我们表中的制造商当前没有生产任何产品，我们可以像下面这样找出这些制造商：

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. `WHERE` 子句

*WHERE 子句*子句的语法是：

```
WHERE _search_condition_
```

这里的 `_search_condition_` 是一个返回类型为 `boolean` 的值表达式(参阅Section 4.2)。

在完成对 `FROM` 子句的处理之后，生成的每一行都会按照搜索条件进行检查。 如果结果是真，那么该行保留在输出表中，否则(也就是结果是假或NULL)就把它抛弃。 搜索条件通常至少要引用一列在 `FROM` 子句里生成的列，这不是必须的， 但如果不这样的话， `WHERE` 子句就没什么意义了。

> **Note:** 内连接的连接条件既可以写在 `WHERE` 子句里也可以写在 `JOIN` 子句里。 比如， 下面的表表达式是等效的：
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> 和：
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> 或者可能还有：
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> 你想用哪个只是风格问题。 `FROM` 子句里的 `JOIN` 语法可能不那么容易移植到其它产品中。即使它是在SQL标准。对于外连接而言， 我们没有选择：连接条件必须在 `FROM` 子句中完成。 外连接的 `ON` 或 `USING` 子句不等于 `WHERE` 条件， 因为它导致最终结果中行的增(那些不匹配的输入行)和删。

这里是一些 `WHERE` 子句的例子：

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

在上面的例子里， `fdt` 是从 `FROM` 子句中派生的表。 那些不符合 `WHERE` 子句的搜索条件的行将从 `fdt` 中删除。 请注意我们把标量子查询当做一个值表达式来用。就像其它查询一样， 子查询里也可以使用复杂的表表达式。同时还请注意子查询 是如何引用 `fdt` 的。把 `c1` 修饰成 `fdt.c1` 只有在 `c1` 是该子查询生成的列名字时才是必须的， 但修饰列名字可以增加语句的准确性(即使有时不是必须的)。 这个例子就演示了字段名字范围如何从外层查询扩展到它的内层查询。

7.2.3. `GROUP BY` 和 `HAVING` 子句

在通过了 `WHERE` 过滤器之后，生成的输入表可以继续用 `GROUP BY` 子句进行分组，然后用 `HAVING` 子句选取一些分组行。

```
SELECT _select_list_
    FROM ...
    [WHERE ...]
    GROUP BY _grouping_column_reference_ [, `_grouping_column_reference_`]...
```

GROUP BY 子句子句用于把那些所有列出的 grouping_column_reference值都相同的行聚集在一起，缩减为一行， 这样就可以删除输出里的重复和/或计算应用于这些组的聚集。这些字段的列出顺序无关紧要。 比如：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT * FROM test1;</kbd>
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT x FROM test1 GROUP BY x;</kbd
 x
---
 a
 b
 c
(3 rows)
```

在第二个查询里，我们不能写成 `SELECT * FROM test1 GROUP BY x`， 因为字段 `y` 里没有哪个值可以和每个组关联起来。 被分组的字段可以在选择列表中引用是因为它们每个组都有单一的数值。

通常，如果一个表被分了组，不在 `GROUP BY` 中列出的字段只能在总表达式中被引用。 一个带聚集表达式的例子是：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT x, sum(y) FROM test1 GROUP BY
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

这里的 `sum` 是一个聚集函数，它在组上计算总和。 有关可用的聚集函数的更多信息可以在Section 9.20中找到。

> **Tip:** 没有有效的聚合表达式分组可以计算一列中不同值的设置。 这个可以通过 `DISTINCT` 子句来实现(参考Section 7.3.3).

这里是另外一个例子：它计算每种产品的总销售额(而不是所有产品的总销售额)。

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

在这个例子里，字段 `product_id`， `p.name` 和 `p.price` 必须在 `GROUP BY` 子句里， 因为它们都在查询选择列表里被引用了（但见下文）。 `s.units` 字段不必在 `GROUP BY` 列表里，因为它只是在一个聚集表达式(`sum(...)`)里使用， 它代表一组产品的销售总额。对于每种产品，这个查询都返回一个该产品的总销售额。

如果产品表是这样设置的，就说 `product_id` 是主键， 那么它足够在上面的例子中对 `product_id` 分组， 因为名字和价格将会函数依赖于产品ID， 这样将不会在返回每个产品ID组时有名字和价格的分歧。

在严格的SQL里， `GROUP BY` 只能对源表的列进行分组，但PostgreSQL 把这个扩展为也允许 `GROUP BY` 对选择列表中的字段进行分组。 也允许对值表达式进行分组，而不仅仅是简单的字段。

如果一个表已经用 `GROUP BY` 分了组， 然后你又只对其中的某些组感兴趣，那么就可以用 `HAVING` 子句筛选分组。 必须像 `WHERE` 子句，从结果中消除组，语法是：

```
SELECT _select_list_ FROM ... [WHERE ...] GROUP BY ... HAVING _boolean_expression_
```

在 `HAVING` 子句中的表达式可以引用分组的表达式和未分组的表达式 (后者必须涉及一个聚集函数)。

例子：

```
<samp class="literal">=></samp> <kbd class="literal">SELECT x, sum(y) FROM test1 GROUP BY
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

<samp class="literal">=></samp> <kbd class="literal">SELECT x, sum(y) FROM test1 GROUP BY
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

然后是一个更现实的例子：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

在上面的例子里， `WHERE` 子句根据未分组的字段选择数据行 (表达式只是对那些最近四周发生的销售为真)。而 `HAVING` 子句在分组之后选择那些销售总额超过5000的组。 请注意聚集表达式不需要在查询中的所有地方都一样。

如果一个查询调用了聚合函数，但没有 `GROUP BY` 子句，分组仍然发生： 结果是单一组行（或者如果单一行被 `HAVING` 所淘汰，那么也许没有行）。 同样，它包含一个 `HAVING` 子句，甚至没有任何聚合函数的调用或 `GROUP BY` 子句。

7.2.4. 窗口函数处理

如果查询包含窗口函数(参考Section 3.5， Section 9.21 和Section 4.2.8)， 这些函数在执行了分组、聚合和 `HAVING` 过滤之后被评估。也就是说，如果查询使用任何的聚合、 `GROUP BY` 或 `HAVING` ， 那么由窗口函数发现的行是该组行而不是从 `FROM` / `WHERE` 得到的原始表行。

当多个窗口函数被使用的时候，所有在它们的窗口定义里依照语法地等效于 `PARTITION BY` 和 `ORDER BY` 子句的窗口函数保证在同一个过去的数据里被评估。 因此它们将看到同样的排序， 即使 `ORDER BY` 不唯一确定一个排序。然而，不确保所做出的关于评价的功能有不同的 `PARTITION BY` 或 `ORDER BY` 规范。 （在这种情况下， 一个排序步骤通常需要在窗口函数评估之间传递，并且不保证行的排序看似跟 `ORDER BY` 等效。）

目前，窗口函数总是需要分类数据，所以查询输出将按照一个或另一个窗口函数的 `PARTITION BY` / `ORDER BY` 子句。它不是说依赖于此。 如果你想要确保结果是按特定的方式分类那么使用显式的顶级ORDER BY子句。

7.3. 选择列表

如前面的小节说明的那样，在 `SELECT` 命令中的表表达式通过组合表、 视图、 删除行、分组等构造了一个中介性的虚拟表。这个表最后传递给选择列表处理。 选择列表判断最终实际输出虚拟表的哪些字段。

7.3.1. 选择列表项

最简单的选择列表是 `*` ，它输出表表达式生成的所有字段。 否则，一个选择列表是一个逗号分隔的值表达式的列表(和在Section 4.2 里定义的一样)。比如，它可能是一个字段名列表：

```
SELECT a, b, c FROM ...
```

字段名 `a` , `b` , `c` 要么是在 `FROM` 子句里引用的表中字段的实际名字，要么是Section 7.2.1.2 里解释的别名。选择列表中的名字空间和 `WHERE` 子句中的名字空间是一样的， 除非你使用了分组，否则它和 `HAVING` 子句中的名字空间也一样。

如果多个表有重复的字段名，那么你还必须给出表名字，例如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

当使用多个表时，给出表名还有助于引用该表的所有字段：

```
SELECT tbl1.*, tbl2.a FROM ...
```

又见Section 7.2.2。

如果将值表达式用于选择列表，那么它在概念上向返回的表中增加了一个新的虚拟字段。 值表达式为结果中的每一行进行一次计算，计算之前用该行的数值替换任何表达式里引用的字段。 不过选择列表中的这个表达式并非一定要引用来自 `FROM` 子句中表表达式里面的字段，比如，它也可以是任意常量算术表达式。

7.3.2. 字段标签

选择列表中的列表项可以赋予名字，以便于进一步的处理。 例如在 `ORDER BY` 子句中的使用或通过客户端应用程序显示。比如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用 `AS` 声明字段名字，那么系统将赋予一个缺省值。 对于简单的字段引用，它是该字段的名字。对于函数调用，它是该函数的名字。 对于复杂表达式，系统会生成一个通用的名字。

只有当新列名与任何PostgreSQL 关键字不匹配时 `AS` 关键字是可选的（见Appendix C）， 您可以给列名加上双引号来避免意外匹配关键字。例如， `VALUE` 是一个关键字， 所以这样是不起作用的：

```
SELECT a value, b + c AS sum FROM ...
```

但这样可以:

```
SELECT a "value", b + c AS sum FROM ...
```

为了防止和未来补充的关键字发生冲突，建议您要么写 `AS` ， 要么为输出列名加双引号标记。

> **Note:** 输出字段的命名和在 `FROM` 子句里的命名是不一样的(参阅Section 7.2.1.2)。 这样就允许你对同一个字段命名两次，FROM 子句里的名字将被选择列表使用， 而选择列表中新取的名字将被最终输出。

7.3.3. `DISTINCT`

在处理完选择列表之后，生成的表可以删除重复行。直接在 `SELECT` 后面写上 `DISTINCT` 关键字即可：

```
SELECT DISTINCT _select_list_ ...
```

如果不用 `DISTINCT` 你可以用 `ALL` 声明保留所有行的缺省行为。

显然，如果两行里至少有一个字段值不同，那么我们认为这两行是独立的。 NULL 在这里被认为是相同的。

另外，我们还可以用表达式来判断什么样的行可以认为是独立的：

```
SELECT DISTINCT ON (_expression_ [, `_expression_` ...]) _select_list_ ...
```

这里的 `_expression_` 是一个值表达式， 它为每一行计算。如果一组行计算出的该表达式的值都相同，那么就认为这些行是重复的， 并只输出第一行。请注意这里的"第一行"是不可预料的， 除非你在足够多的字段上对该查询进行了排序，保证到达 `DISTINCT` 过滤器时行的顺序是唯一的(`DISTINCT ON` 将在 `ORDER BY` 排序之后处理)。

`DISTINCT ON`子句不是 SQL 标准的一部分，有时候被认为是一个糟糕的风格， 因为它的结果是不可判定的。如果用有可选的`GROUP BY`和在`FROM` 中的子查询可以达到目的，那么我们可以避免使用这个构造，但是通常它是更方便的方法。

7.4. 组合查询

可以对两个查询的结果进行集合操作(并、交、差)。语法是：

```
_query1_ UNION [ALL] _query2_
_query1_ INTERSECT [ALL] _query2_
_query1_ EXCEPT [ALL] _query2_
```

`_query1_` 和 `_query2_` 可以是讨论过的所有查询。集合操作也可以嵌套和级连，比如：

```
_query1_ UNION _query2_ UNION _query3_
```

它实际上等价于：

```
(_query1_ UNION _query2_) UNION _query3_
```

`UNION` 把 `_query2_` 的结果附加到 `_query1_` 的结果上(不过我们不能保证这就是这些行实际的返回顺序)，并且像 `DISTINCT` 那样删除结果中所有重复的行(除非声明了 `UNION ALL`)。

`INTERSECT` 返回那些同时存在于 `_query1_` 和 `_query2_` 结果中的行，除非声明了 `INTERSECT ALL` ， 否则所有重复行都被删除。

`EXCEPT` 返回所有在 `_query1_` 结果中但是不在 `_query2_` 结果中的行(有时侯这叫做两个查询的差)。除非声明了 `EXCEPT ALL` ， 否则所有重复行都被删除。

为了能够计算两个查询的并、交、差，这两个查询必须是"并集兼容的"， 也就是它们都返回同样数量的列，并且对应的列有兼容的数据类型， 就像Section 10.5里描述的那样。

7.5. 行排序

在查询生成输出表之后，也就是在处理完选择列表之后，你还可以对输出表进行排序。 如果没有排序，那么行将以不可预测的顺序返回(实际顺序将取决于扫描和连接规划类型和在磁盘上的顺序， 但是肯定不能依赖这些东西)。确定的顺序只能在明确地使用了排序步骤之后才能保证。

`ORDER BY` 子句用于声明排序顺序:

```
SELECT _select_list_
    FROM _table_expression_
    ORDER BY _sort_expression1_ [ASC | DESC] [NULLS { FIRST | LAST }]
             [, `_sort_expression2_` [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

sort_expression 是任何可用于选择列表的表达式，例如：

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

如果指定了多个排序表达式，那么仅在前面的表达式排序相等的情况下才使用后面的表达式做进一步排序。 每个表达式都可以跟一个可选的 `ASC` (升序，默认) 或 `DESC` (降序)以设置排序方向。 升序先输出小的数值，这里的"小"是以 `<` 操作符的角度定义的。 类似的是，降序是以 `>` 操作符来判断的。 [1]

`NULLS FIRST` 和 `NULLS LAST` 选项可以决定在排序操作中在 non-null 值之前还是之后。默认情况下，空值大于任何非空值；也就是说， `DESC` 排序默认是 `NULLS FIRST` ， 否则为 `NULLS LAST` 。

注意，排序选项对于每个排序列是相对独立的。例如 `ORDER BY x, y DESC` 意思是说 `ORDER BY x ASC, y DESC` ，不同于 `ORDER BY x DESC, y DESC` 。

一个 `_sort_expression_` 也可以是字段名或字段编号，如:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

都按照第一个字段进行排序。需要注意的是，输出字段名必须是独立的(不允许在表达式中使用)。 比如，下面的语句是错误的：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- 错误的
```

这样的限制主要是为了避免歧义。另外，如果某个排序表达式能够同时匹配输出字段名和表表达式中的字段名，也会导致歧义(此时使用输出字段名)。当然，这种情况仅在你使用了`AS`重命名输出字段并且恰好与其它表的字段同名的时候才会发生。

`ORDER BY`可以应用于`UNION`，`INTERSECT`，`EXCEPT`组合的计算结果，不过在这种情况下，只允许按照字段的名字或编号进行排序，而不允许按照表达式进行排序。

Notes

[1] 事实上，PostgreSQL使用默认的*B-tree*操作符类为表达式的数据类型确定`ASC`和`DESC`排序顺序。一般来说，数据类型将被转换为适合于`<`和`>`操作符进行排序。但是对于用户自定义的数据类型可以不必如此。

7.6. `LIMIT` 和 `OFFSET`

`LIMIT` 和 `OFFSET` 子句允许你只取出查询结果中的一部分数据行：

```
SELECT _select_list_
    FROM _table_expression_
    [ ORDER BY ... ]
    [ LIMIT { `_number_` | ALL } ] [ OFFSET `_number_` ]
```

如果给出了一个 LIMIT 计数，那么将返回不超过该数字的行(也可能更少些， 因为可能查询本身生成的总行数就比较少)。 `LIMIT ALL` 和省略 `LIMIT` 子句是一样的。

`OFFSET` 指明在开始返回行之前忽略多少行。 `OFFSET 0` 和省略 `OFFSET` 子句是一样的， `LIMIT NULL` 和省略 `LIMIT` 子句 是一样的。如果 `OFFSET` 和 `LIMIT` 都出现了，那么在计算返回的 `LIMIT` 之前先忽略 `OFFSET` 指定的行数。

使用 `LIMIT` 的同时使用 `ORDER BY` 子句把结果行约束成一个唯一的顺序是一个好主意。 否则你就会得到一个不可预料的子集。你要的可能是第十到二十行，但以什么顺序的十到二十？ 除非你声明了 `ORDER BY` ， 否则顺序是未知的。

查询优化器在生成查询规划的时候会考虑 `LIMIT` ， 因此如果你给 `LIMIT` 和 `OFFSET` 的值不同， 那么你很可能得到不同的规划(产生不同的行顺序)。因此， 使用不同的 `LIMIT` / `OFFSET` 值选择不同的子集将生成不一致的结果， 除非你用 `ORDER BY` 强制一个可预料的顺序。这可不是臭虫， 而是一个很自然的结果，因为 SQL 没有许诺把查询的结果按照任何特定的顺序发出， 除非用了 `ORDER BY` 来约束顺序。

`OFFSET` 子句忽略的行仍然需要在服务器内部计算； 因此，一个很大的 `OFFSET` 可能还是不够有效率。

7.7. `VALUES` 列表

可以在查询中使用由 `VALUES` 生成的"常数表"， 而无需在磁盘上实际创建这个表。语法如下：

```
VALUES ( _expression_ [, ...] ) [, ...]
```

每个括号中的表达式列表生成表中的一行。每个列表中的项数(也就是字段数)必须相等， 并且对应的数据类型必须兼容。最终表中每个字段的数据类型将使用与 `UNION` (参见Section 10.5)相同的规则确定。

例如:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

将得到 2 列 3 行的表。并且与下面的语句等价：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

PostgreSQL默认将 `VALUES` 所得到的表中各字段分别命名为 `column1` , `column2` 等等。 SQL标准并未规定此种情况下的字段名命名规范，不同的数据库系统对此的处理也各不相同， 所以最好明确指定字段的名字，像下面这样：

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

语法上，带有表达式列表的 `VALUES` 和下面的语句等价：

```
SELECT _select_list_ FROM _table_expression_
```

并且可以出现在任何 `SELECT` 可以出现的地方。例如，你可以把它用于 `UNION` 的一部分，或者在其上附加一个 `_sort_specification_` (`ORDER BY` , `LIMIT` , `OFFSET`)。 `VALUES` 通常用作 `INSERT` 命令的数据源或者子查询。

更多信息参见VALUES。

7.8. `WITH` 查询 (通用表表达式)

`WITH` 提供了一种在更大的查询中编写辅助语句的方式。 这个通常称为通用表表达式或CTEs的辅助语句可以认为是定义只存在于一个查询中的临时表。 每个 `WITH` 子句中的辅助语句可以是一个 `SELECT` , `INSERT` , `UPDATE` 或 `DELETE` ；并且 `WITH` 子句本身附加到的初级语句可以是一个 `SELECT` , `INSERT` , `UPDATE` 或 `DELETE` 。

7.8.1. `WITH` 中的 `SELECT`

`WITH` 中 `SELECT` 的本意是为了将复杂的查询分解为更简单的部分。一个例子是：

```
WITH regional_sales AS (
        SELECT region, SUM(amount) AS total_sales
        FROM orders
        GROUP BY region
     ), top_regions AS (
        SELECT region
        FROM regional_sales
        WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
     )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

它显示了每个产品仅在销售区域的销售总额。 `WITH` 子句定义了两个名为 `regional_sales` 和 `top_regions` 的辅助语句， `regional_sales` 的输出用于 `top_regions` ， 而 `top_regions` 的输出用于初级的 `SELECT` 查询。 这个例子也可以不用 `WITH` 来写，但是需要两级嵌套的子 `SELECT` 查询。 用这种方法更容易理解。

可选的 `RECURSIVE` 修饰符将 `WITH` 从一个单纯的语法方便改变为在SQL标准中不可能实现的功能。 使用 `RECURSIVE` ，一个 `WITH` 查询可以引用它自己的输出。 一个非常简单的例子是查询1到100的和：

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

一个递归 `WITH` 查询的一般形式总是一个*non-recursive term*， 然后 `UNION` （或者 `UNION ALL` ）， 然后一个*recursive term*， 其中只有递归的术语可以包含一个对查询自己输出的引用。这样一个查询像下面那样执行：

递归查询评估

1. 评估非递归的术语。使用 `UNION` （而不是 `UNION ALL` ）去除重复的行。 包括在递归查询结果中所有剩余的行，并将它们放入临时的工作表。

2. 只要工作表不为空，那么将重复这些步骤：

   i. 评估递归术语，为递归自我参照替换当前工作表内容。用 `UNION` (并不是 `UNION ALL`)， 去除重复的行和与以前结果行重复的行。包括所有在递归查询结果中剩余的行， 并将它们放入一个临时的中间表。

   ii. 用中间表的内容替换工作表的内容，然后清空中间表。

> **Note:** 严格的说，该过程是迭代而不是递归，但是 `RECURSIVE` 是通过 SQL 标准委员会选择的术语。

在上面的例子中，在每一步中仅有一个工作表行，并且在后续的步骤中它的值将从 1 升至 100。 在第 100 步，因为 `WHERE` 子句的原因没有任何输出，因此查询终止。

递归查询通常用于处理分层或树状结构数据。一个有用的示例查询是查找所有直接或间接的产品的附带部分， 仅提供一个表来显示即时的包含：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

当使用递归查询的时候，确保查询的递归部分最终不会返回元组是很重要的， 否则查询将会无限的循环下去。有时，通过使用 `UNION` 替代 `UNION ALL` 去除掉与前面输出重复的行可以实现这个。然而，通常一个周期不涉及那些完全复制的输出行： 检查一个或几个字段来查看是否存在事先达成的相同点可能是必要的。 处理这种情况的标准方式是计算一个已经访问过的数值的数组。 例如，请考虑下面的查询，使用 `link` 字段搜索一个表 `graph` ：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果 `link` 关系包含循环那么这个查询将会循环。 因为我们需要一个"深度"输出，仅改变 `UNION ALL` 为 `UNION` 将不会消除循环。相反，我们需要认识到当我们按照特定的链接路径时是否再次得到了相同的行。 我们添加两列 `path` 和 `cycle` 到倾向循环的查询：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[g.id],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || g.id,
          g.id = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了防止循环，该数组值通常是有用的，在它的右边作为代表用来得到任何特定行的"路径"。

在一般情况下，需要检测多个字段来识别一个循环时使用一个行数组。例如， 如果我们需要对比字段 `f1` 和 `f2` ：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **Tip:** 在常见的情况下，当只需要检查一个字段来识别循环的时候忽略 `ROW()` 语法。 这允许使用一个简单的数组而不是一个复杂类型的数组，增加查询的效率。
>
> **Tip:** 递归查询评估算法产生以广度优先搜索顺序的输出。 您可以按照深度优先查询排序通过外部查询 `ORDER BY` 一个"path"列来显示结果。

当您不能确定它们是否会循环的时候，在一个父查询中放置 `LIMIT` 是一个对于测试查询有用的技巧。 例如，这个查询将在没有 `LIMIT` 的情况下无限循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

它能工作是因为PostgreSQL 的实现评估只有 `WITH` 查询的行实际上是通过父查询获取的。 在实际的生产环境下不推荐使用该技巧，因为其它的系统可能以不同的方式工作。 同样，如果您使用外部查询将递归查询结果排序或将它们加入到别的表中， 那么它通常是不工作的，因为在这种情况下外部查询将获取所有 `WITH` 查询的输出。

一个有用的 `WITH` 查询属性是每个父查询执行一次它们只做一次评估， 即使它们不止一次地通过父查询或 `WITH` 查询引用。所以， 昂贵的需要在多个地方放置的计算可以通过设置 `WITH` 查询来避免冗余工作。 另一个可能的应用是防止不必要的副作用函数的多个评估。 然而，另一方面，比起普通的子查询， 优化器不能够避开父查询拆分为一个 `WITH` 查询的限制。通常 `WITH` 查询将如上评估，没有行限制的父查询可能丢失。（但是，正如上面所说，如果查询参考只需要数量有限的行，评估可能会很早终止。）

上面的例子只显示了 `WITH` 在 `SELECT` 中的使用， 但是它也可以用同样的方式附加到 `INSERT`， `UPDATE` 或 `DELETE` 。 在每种情况下它都有效的提供可以在主要的命令中引用的临时表。

7.8.2. `WITH` 中的数据修改语句

你可以在 `WITH` 中使用数据修改语句(`INSERT` , `UPDATE` 或 `DELETE`)。这允许你在相同的查询中执行几个不同的操作，一个例子是：

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

这个查询有效的移动 `products` 中的行到 `products_log` 。 `WITH` 中的 `DELETE` 从 `products` 中删除指定的行， 并且通过 `RETURNING` 子句返回它们的内容； 然后初级查询读取那个输出并且插入到 `products_log` 中。

上面例子的一个优点是 `WITH` 子句是附加到 `INSERT` ， 而不是 `INSERT` 中的子 `SELECT` 查询。 这是必须的，因为数据修改语句只允许在附加到顶级语句的 `WITH` 子句中使用。 然而，因为正常的 `WITH` 可见性规则的应用，所以从子 `SELECT` 查询中引用 `WITH` 语句的输出是可能的。

在 `WITH` 中的数据修改语句通常都有 `RETURNING` 子句，就像上面的例子一样。 它是 `RETURNING` 子句的输出，不是数据修改语句的目标表， 形成的临时表可以被其他的查询引用。如果 `WITH` 中的数据修改语句缺少了 `RETURNING` 子句，那么将没有临时表生成，也就不能被其他的查询引用。 这样的语句将仍然被执行。一个不是特别有用的例子是：

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

这个例子将删除表 `foo` 和 `bar` 中的所有行。 报告给客户端的受影响行的数量将只包含从 `bar` 中删除的行。

数据修改语句中不允许递归的自引用。在某些情况下通过引用递归的 `WITH` 输出，可能绕开这个限制，例如：

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

这个查询将删除一个产品所有直接或非直接的subparts。

`WITH` 中的数据修改语句被直接执行一次，并且总是完成， 独立的主查询读取所有（或者实际上是任意）它们的输出。 注意，这和在 `WITH` 中 `SELECT` 的规则不同： 就像前一节规定的那样， `SELECT` 的执行直到首级查询需要它的输出时才实施。

`WITH` 中的子语句之间和与主查询之间兼容的执行。因此， 当在 `WITH` 中使用数据修改语句时，其他的指定的更新实际上是不可预知发生的。 所有的语句都在相同的快照中执行（见 Chapter 13）， 所以他们不能"看见"彼此对目标表的影响。这样减轻了实际行更新的不可预知的影响， 并且意味着 `RETURNING` 数据是唯一在不同的 `WITH` 子语句和主查询间交流变化的方式。 一个例子是：

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

外层的 `SELECT` 将在 `UPDATE` 动作之前返回原价，而在：

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

中，外层 `SELECT` 将返回更新了的数据。

不支持尝试在一个语句中更新相同的行两次。如果尝试了，那么只有一个修改会发生， 但是不容易（或者有时不可能）准确预测是哪一个。这个同样适用于删除一个已经在相同语句中更新了的行：只有更新被执行。因此你通常应该避免尝试在一个语句中修改一个行两次。特别的， 避免写可能影响被主语句或同级子语句改变了的行的 `WITH` 子语句。 这样一个语句的影响将是不可预测的。

目前，任何作为在 `WITH` 中的数据修改语句目标的表，不必有扩展到多个语句的条件规则、 `ALSO` 规则和 `INSTEAD` 规则。

Chapter 8. 数据类型

Table of Contents

PostgreSQL有着丰富的内置数据类型可用。 用户还可以使用CREATE TYPE命令增加新的数据类型。

Table 8-1显示了所有内置的普通数据类型。 在"别名"列里列出的大多数可选名字都是因历史原因PostgreSQL 在内部使用的名字。 另外， 还有一些内部使用的或者废弃的类型也可以使用， 但没有在这里列出。

Table 8-1. 数据类型

名字	别名	描述
bigint	int8	有符号8字节整数
bigserial	serial8	自增8字节整数
bit [(``_n_)]	定长位串	
bit varying [(``_n_)]	varbit	变长位串
boolean	bool	逻辑布尔值(真/假)
box	平面上的矩形	
bytea	二进制数据("字节数组")	
character varying [(``_n_)]	varchar [(``_n_)]	变长字符串
character [(``_n_)]	char [(``_n_)]	定长字符串
cidr	IPv4 或 IPv6 网络地址	
circle	平面上的圆	
date	日历日期(年, 月, 日)	
double precision	float8	双精度浮点数字(8字节)
inet	IPv4 或 IPv6 主机地址	
integer	int , int4	有符号 4 字节整数
interval [_fields_] [(_p_)]	时间间隔	
line	平面上的无限长直线	
lseg	平面上的线段	
macaddr	MAC (Media Access Control)地址	
money	货币金额	
numeric [(``_p_ , _s_)]	decimal [(``_p_ , _s_)]	可选精度的准确数字
path	平面上的几何路径	
point	平面上的点	
polygon	平面上的封闭几何路径	
		单精度浮点数(4 字

		节)
smallint	int2	有符号 2 字节整数
smallserial	serial2	自增 2 字节整数
serial	serial4	自增 4 字节整数
text	变长字符串	
time [(``_p_)] [without time zone]	一天中的时间(无时区)	
time [(``_p_)] with time zone	timetz	一天里的时间，包括时区
timestamp [(``_p_)] [without time zone]	日期和时间(无时区)	
timestamp [(``_p_)] with time zone	timestamptz	日期和时间，包括时区
tsquery	文本检索查询	
tsvector	文本检索文档	
txid_snapshot	用户级别的事务ID快照	
uuid	通用唯一标识符	
xml	XML 数据	
json	JSON 数据	

> 兼容性: 下列类型(或者那样拼写的)是SQL声明的： bigint , bit , bit varying , boolean , char , character varying , character , varchar , date , double precision , integer , interval , numeric , decimal , real , smallint , time (有时区和无时区), timestamp (有时区和无时区), xml 。

每种数据类型都有一个由其输入和输出函数决定的外部表现形式。 许多内建的类型有明显的格式。不过，许多类型要么是PostgreSQL 所特有的，比如几何路径，要么是有几种不同的格式，比如日期和时间类型。 有些输入和输出函数是不可逆的。也就是说， 输出函数的输出结果和原始的输入比较的时候可能丢失精度。

8.1. 数值类型

数值类型由 2、4 或 8 字节的整数以及 4 或 8 字节的浮点数和可选精度的小数组成。 Table 8-2列出了所有可用类型。

Table 8-2. 数值类型

名字	存储空间	描述	范围
`smallint`	2 字节	小范围整数	-32768 到 +32767
`integer`	4 字节	常用的整数	-2147483648 到 +2147483647
`bigint`	8 字节	大范围整数	-9223372036854775808 到 +9223372036854775807
`decimal`	变长	用户声明精度，精确	小数点前 131072 位；小数点后 16383 位
`numeric`	变长	用户声明精度，精确	小数点前 131072 位；小数点后 16383 位
`real`	4 字节	变精度，不精确	6 位十进制数字精度
`double precision`	8 字节	变精度，不精确	15 位十进制数字精度
`smallserial`	2 字节	小范围自增整数	1 到 32767
`serial`	4 字节	自增整数	1 到 2147483647
`bigserial`	8 字节	大范围自增整数	1 到 9223372036854775807

数值类型常量的语法在Section 4.1.2里描述。 数值类型对应有一套完整的数学操作符和函数。相关信息请参考Chapter 9。 下面的几节详细描述这些类型。

8.1.1. 整数类型

`smallint`， `integer` 和 `bigint` 类型存储各种范围的全部是数字的数， 也就是没有小数部分的数字。试图存储超出范围以外的数值将导致一个错误。

常用的类型是 `integer` ，因为它提供了在范围、存储空间、性能之间的最佳平衡。 一般只有在磁盘空间紧张的时候才使用 `smallint` 。 当 `integer` 的范围不够的时候才使用 `bigint` 。

SQL只声明了整数类型 `integer` (或 `int`)， `smallint` 和 `bigint` 。类型 `int2` , `int4` 和 `int8` 都是扩展，并且也在许多其它SQL数据库系统中使用。

8.1.2. 任意精度数值

`numeric` 类型可以存储非常大的数字并且准确地进行计算。 我们特别建议将它用于货币金额和其它要求精确计算的场合。不过， `numeric` 类型上的算术运算比整数类型或者我们下一节描述的浮点数类型要慢很多。

在随后的内容里，我们使用下述术语：一个 `numeric` 类型的标度 (*scale*)是小数部分的位数，精度(*precision*) 是全部数据位的数目，也就是小数点两边的位数总和。 因此数字 23.5141 的精度为 6 而标度为 4 。你可以认为整数的标度为零。

`numeric` 字段的最大精度和最大标度都是可以配置的。 要声明一个字段的类型为 `numeric` ，你可以用下面的语法：

```
NUMERIC(_precision_, _scale_)
```

精度必须为正数，标度可以为零或者正数。另外:

```
NUMERIC(_precision_)
```

选择了标度为 0 。不带任何精度与标度的声明

```
NUMERIC
```

则创建一个可以存储一个直到实现精度上限的任意精度和标度的数值， 一个这样类型的字段将不会把输入数值转化成任何特定的标度， 而带有标度声明的 `numeric` 字段将把输入值转化为该标度。 SQL标准要求缺省的标度是 0(也就是转化成整数精度)。 我们觉得这样做有点没用。如果你关心移植性， 那你最好总是明确声明精度和标度。

> **Note:** 当在类型声明中显示指定精度时允许的最大值为1000；没有指定精度的 `NUMERIC` 遵从Table 8-2里的描述。

如果一个要存储的数值的标度比字段声明的标度高， 那么系统将尝试圆整(四舍五入)该数值到指定的小数位。然后， 如果小数点左边的数据位数超过了声明的精度减去声明的标度，那么将抛出一个错误。

numeric 类型的数据值在物理上是不带任何前导或者后缀零的形式存储的。 因此，字段上声明的精度和标度都是最大值，而不是固定分配的。在这个方面， `numeric` 类型更类似于 `varchar(``_n_` )而不是 `char(``_n_`)。实际存储是每四个十进制位两个字节， 然后在整个数据上加上三到八个字节的额外开销。

除了普通的数字值之外， `numeric` 类型允许用特殊值 `NaN` 表示"不是一个数字"。任何在 `NaN` 上面的操作都生成另外一个 `NaN` 。如果在 SQL 命令里把这些值当作一个常量写， 你必须在其周围放上单引号，比如 `UPDATE table SET x = 'NaN'` 。 在输入时，字符串 `NaN` 是大小写无关的。

> **Note:** 在大多数"not-a-number"概念中，不认为 `NaN` 等于其他数值类型 （包括 `NaN` ）。 为了能够存储 `numeric` 类型的值， 并且使用Tree索引，PostgreSQL认为 `NaN` 相等， 并且大于所有非 `NaN` 值。

类型 `decimal` 和 `numeric` 是等效的。 两种类型都是SQL标准。

8.1.3. 浮点数类型

数据类型 `real` 和 `double precision` 是不精确的、变精度的数字类型。 实际上，这些类型是IEEE 754标准二进制浮点数算术(分别对应单和双精度)的一般实现， 外加下层处理器、操作系统和编译器对它的支持。

不精确意味着一些数值不能精确地转换成内部格式并且是以近似值存储的， 因此存储后再把数据打印出来可能有一些差异。 处理这些错误以及这些错误是如何在计算中传播的属于数学和计算机科学的一个完整的分支， 我们不会在这里进一步讨论它，这里的讨论仅限于如下几点：

- 如果你要求精确的计算(比如计算货币金额)， 应使用 `numeric` 类型。
- 如果你想用这些类型做任何重要的复杂计算， 尤其是那些你对范围情况(无穷/下溢)严重依赖的事情，那你应该仔细评估你的实现。
- 拿两个浮点数值进行相等性比较可能不像你想像那样运转。

在大多数平台上， `real` 类型的范围是至少1E-37到1E+37， 精度至少是6位小数。 `double precision` 的范围通常是1E-307到1E+308， 精度是至少15位数字。太大或者太小的数值都会导致错误。如果输入数据的精度太高， 那么将会发生圆整。太接近零的数字，如果无法与零值的表现形式相区分就会产生下溢错误。

> **Note:** 当一个浮点数值转化为文本输出时，extra_float_digits 设置控制额外有效数字的位数。默认值是 `0` ，PostgreSQL 支持的平台上的输出是一样的。增加这个值产生的输出将更精确的表示存储值， 但是可能不利于移植。

除了普通的数字值之外，浮点类型还有几个特殊值：

`Infinity` `-Infinity` `NaN`

这些值分别表示 IEEE 754 特殊值"正无穷大"、"负无穷大"、 "不是一个数字"。在不遵循 IEEE 754 浮点算术的机器上， 这些值的含义可能不是预期的。如果在 SQL 命令里把这些数值当作常量写， 你必须在它们周围放上单引号，像这样： `UPDATE table SET x = 'Infinity'` 。 输入时，这些值是以大小写无关的方式识别的。

> **Note:** IEEE754声明 `NaN` 不应该等于任何其他浮点值（包括 `NaN` ）。 为了能存储浮点值，并且使用Tree索引，PostgreSQL认为 `NaN` 相等，并且大于所有非 `NaN` 值。

PostgreSQL还支持 SQL 标准表示法 `float` 和 `float(``_p_` )用于声明非精确的数值类型。 其中的 `_p_` 声明以二进制位表示的最低可接受精度。 在选取 `real` 类型的时候，PostgreSQL接受 `float(1)` 到 `float(24)` ，在选取 `double precision` 的时候，接受 `float(25)` 到 `float(53)` 。 在允许范围之外的 `_p_` 值将导致一个错误。 没有声明精度的 `float` 将被当作 `double precision` 。

> **Note:** PostgreSQL 7.4以前，在 `float(``_p_` ) 里面的精度会被当作是这么多位数的十进制位。到 7.4 已经被修改成与 SQL 标准匹配， 标准声明这个精度是以二进制位度量的。假设 `real` 和 `double precision` 分别有 24 和 53 个二进制位的位数对 IEEE 标准的浮点实现来说是正确的。在非 IEEE 平台上， 这个数值可能略有偏差，但是为了简化，我们在所有平台上都用了同样的 `_p_` 值范围。

8.1.4. 序列号类型

`smallserial` , `serial` 和 `bigserial` 类型不是真正的类型， 只是为在表中创建唯一标识做的概念上的便利。类似其它一些数据库中的 `AUTO_INCREMENT` 属性。在目前的实现中，下面一个语句：

```
CREATE TABLE _tablename_ (
    _colname_ SERIAL
);
```

等价于声明下面几个语句：

```
CREATE SEQUENCE _tablename___colname__seq;
CREATE TABLE _tablename_ (
    _colname_ integer NOT NULL DEFAULT nextval('_tablename___colname__seq')
);
ALTER SEQUENCE _tablename___colname__seq OWNED BY _tablename_._colname_;
```

因此，我们就创建了一个整数字段并且把它的缺省数值安排为从一个序列发生器读取。 应用了一个 `NOT NULL` 约束以确保 NULL 不会被插入。 在大多数情况下你可能还希望附加一个 `UNIQUE` 或 `PRIMARY KEY` 约束避免意外地插入重复的数值，但这个不是自动的。最后， 将序列发生器"从属于"那个字段，这样当该字段或表被删除的时候也一并删除它。

> **Note:** 因为 `smallserial`，`serial` 和 `bigserial` 是使用序列实现的，所以显示在字段里的序列值可能有“漏洞”或者缺口，即使没有列曾经被删除。 一个从序列中分配的值仍然会“使用”，即使包含这个值的行没有成功的插入到表格的字段中。 这种情况是有可能发生的，比如，插入事务回滚。参阅Section 9.16 中的 `nextval()` 获取详细信息。
>
> **Note:** PostgreSQL 7.3以前， `serial` 隐含 `UNIQUE` 。 但现在不再如此。如果你希望一个序列字段有一个唯一约束或者一个主键， 那么你现在必须声明，就像其它数据类型一样。

要在 `serial` 字段中插入序列中的下一个数值，主要是要注意 `serial` 字段应该赋予缺省值。 我们可以通过在 `INSERT` 语句中把该字段排除在字段列表之外来实现，也可以通过使用 `DEFAULT` 关键字来实现。

类型名 `serial` 和 `serial4` 是等效的： 两者都创建 `integer` 字段。类型名 `bigserial` 和 `serial8` 也一样，只不过它创建一个 `bigint` 字段。 如果你预计在表的生存期中使用的标识数目可能超过 2³¹ 个， 那么你应该使用 `bigserial` 。类型名 `smallserial` 和 `serial2` 也一样，只不过它创建一个 `smallint` 字段。

一个 `serial` 类型创建的序列在所属的字段被删除的时候自动删除。 你可以只删除序列而不删除字段，不过这将删除该字段的缺省值表达式。

8.2. 货币类型

`money` 类型存储带有固定小数精度的货币金额，可查阅 Table 8-3。小数精度由lc_monetary 的设置来决定。表格中显示的范围假设有两位小数。可以以任意格式输入，包括整型， 浮点型，或者典型的货币格式，如 `'$1,000.00'` 。 根据区域字符集，输出一般是最后一种形式。

Table 8-3. 货币类型

名字	存储容量	描述	范围
money	8 字节	货币金额	-92233720368547758.08 到 +92233720368547758.07

由于输出的数据类型对语言环境要求很细，因此， `lc_monetary` 设置的不同可能会造成无法将 `money` 数据输入到数据库中。 为了避免这种问题的发生，在向一个新数据库进行转储之前，确保 `lc_monetary` 与原数据库相同，或具有等价值。

`numeric` ， `int` 和 `bigint` 数据类型的值可以转化为 `money` 类型。从 `real` 和 `double precision` 数据类型的转换可以通过先转化为 `numeric` 类型，例如：

```
SELECT '12.34'::float8::numeric::money;
```

然而，这是不被建议的。浮点数不应该用来处理货币类型，因为潜在的圆整可能导致错误。

`money` 值可以被转换为 `numeric` 而不丢失精度。 转换为其他类型可能丢失精度，并且必须通过两步来完成：

```
SELECT '52093.89'::money::numeric::float8;
```

当一个 `money` 值被另一个 `money` 值除时， 结果是 `double precision` （也就是，一个纯数字，而不是money） ； 在结果中货币单位相互取消。

8.3. 字符类型

Table 8-4. 字符类型

名字	描述
`character varying(``_n_)`, `varchar(``_n_)`	变长，有长度限制
`character(``_n_)`, `char(``_n_)`	定长，不足补空白
`text`	变长，无长度限制

Table 8-4显示了在PostgreSQL 里可用于一般用途的字符类型。

SQL定义了两种基本的字符类型： `character varying(``_n_)` 和 `character(``_n_)`， 这里的 `_n_` 是一个正整数。两种类型都可以存储最多 `_n_` 个字符的字符串（没有字节）。 试图存储更长的字符串到这些类型的字段里会产生一个错误， 除非超出长度的字符都是空白，这种情况下该字符串将被截断为最大长度。 这个看上去有点怪异的例外是SQL标准要求的。 如果要存储的字符串比声明的长度短，类型为 `character` 的数值将会用空白填满； 而类型为 `character varying` 的数值将只是存储短些的字符串。

如果我们明确地把一个数值转换成 `character varying(``_n_)` 或 `character(``_n_)`，那么超长的数值将被截断成 `_n_` 个字符，且不会抛出错误。这也是SQL标准的要求。

`varchar(``_n_)`和 `char(``_n_)` 分别是 `character varying(``_n_)` 和 `character(``_n_)`的别名， 没有声明长度的 `character` 等于 `character(1)` ；如果不带长度说明词使用 `character varying` ， 那么该类型接受任何长度的字符串。后者是PostgreSQL的扩展。

另外，PostgreSQL提供 `text` 类型， 它可以存储任何长度的字符串。尽管类型 `text` 不是SQL 标准，但是许多其它SQL数据库系统也有它。

`character` 类型的数值物理上都用空白填充到指定的长度 `_n_`， 并且以这种方式存储和显示。不过，填充的空白是无语意的。在比较两个 `character` 值的时候，填充的空白都不会被关注，在转换成其它字符串类型的时候， `character` 值里面的空白会被删除。请注意，在 `character varying` 和 `text` 数值里，结尾的空白是有语意的。 并且当使用模式匹配时，如 `LIKE` ，使用正则表达式。

一个简短的字符串（最多126个字节）的存储要求是1个字节加上实际的字符串， 其中包括空格填充的 `character` 。更长的字符串有4个字节的开销， 而不是1。长的字符串将会自动被系统压缩，因此在磁盘上的物理需求可能会更少些。 更长的数值也会存储在后台表里面，这样它们就不会干扰对短字段值的快速访问。 不管怎样，允许存储的最长字符串大概是1GB 。允许在数据类型声明中出现的 `_n_` 的最大值比这还小。修改这个行为没有什么意义，因为在多字节编码下字符和字节的数目可能差别很大。 如果你想存储没有特定上限的长字符串，那么使用 `text` 或没有长度声明的 `character varying` ，而不要选择一个任意长度限制。

> **Tip:** 这三种类型之间没有性能差别，除了当使用填充空白类型时的增加存储空间， 和当存储长度约束的列时一些检查存入时长度的额外的CPU周期。 虽然在某些其它的数据库系统里， `character(``_n_`) 有一定的性能优势，但在PostgreSQL里没有。 事实上， `character(``_n_`)通常是这三个中最慢的， 因为额外存储成本。在大多数情况下， 应该使用 `text` 或 `character varying` 。

请参考Section 4.1.2.1获取关于字符串文本的语法的信息， 以及Chapter 9获取关于可用操作符和函数的信息。 数据库的字符集决定用于存储文本值的字符集；有关字符集支持的更多信息， 请参考Section 22.3。

Example 8-1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- <a name="CO.DATATYPE-CHAR">**(1)**</a>
 <samp class="literal">a    | char_length
------+-------------
 ok   |           2</samp>

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
<samp class="literal">ERROR:  value too long for type character varying(5)</samp>
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- 明确截断
SELECT b, char_length(b) FROM test2;
 <samp class="literal">b     | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5</samp>
```

(1)

`char_length` 函数在Section 9.4中讨论。

在PostgreSQL里另外还有两种定长字符类型。 在Table 8-5里显示。 `name` 类型只用于在内部系统表中存储标识符并且不是给一般用户使用的。 该类型长度当前定为 64 字节(63 可用字符加结束符)但应该使用 `C` 源码中的常量 `NAMEDATALEN` 引用。这个长度是在编译的时候设置的，因而可以为特殊用途调整，缺省的最大长度在以后的版本可能会改变。 类型 `"char"` (注意引号)和 `char(1)` 是不一样的， 它只用了一个字节的存储空间。它在系统内部用于系统表当做过分简单化的枚举类型用。

Table 8-5. 特殊字符类型

名字	存储空间	描述
`"char"`	1 字节	单字节内部类型
`name`	64 字节	用于对象名的内部类型

8.4. 二进制数据类型

`bytea` 数据类型允许存储二进制字符串。参阅Table 8-6。

Table 8-6. 二进制数据类型

名字	存储空间	描述
`bytea`	1或4字节加上实际的二进制字符串	变长的二进制字符串

二进制字符串是一个字节序列。二进制字符串和普通字符字符串的区别有两个： 首先，二进制字符串完全可以存储字节零值以及其它"不可打印的" 字节(定义在 32 到 126 范围之外的字节)。字符串不允许字节零值， 并且也不允许那些不符合选定的字符集编码的非法字节值或者字节序列。 第二，对二进制字符串的处理实际上就是处理字节，而对字符串的处理则取决于区域设置。 简单说，二进制字符串适用于存储那些程序员认为是"原始字节"的数据， 而字符串适合存储文本。

`bytea` 类型支持两种输入输出的外部格式："hex" 格式和PostgreSQL的历史"escape"格式。这两种格式通常在输入中使用，输出格式由bytea_output 配置参数决定，默认是hex。（需要注意的是hex格式是在PostgreSQL 9.0中引入的，早期版本和一些工具中不识别。）

SQL标准定义了一个不同的二进制字符串格式，称为 `BLOB` 或 `BINARY LARGE OBJECT` 。输入格式与 `bytea` 不同， 但提供的函数和操作符基本一致。

8.4.1. `bytea` 十六进制格式

"hex"格式将二进制数据编码为每字节2位十六进制数字， 最重要的四位（半字节）放在开始。整条字符串以 `\x` 开始 （与逃逸格式相区别）。在某些情况下，最初的反斜杠需要再写一次，以逃逸， 同样，在逃逸格式中，反斜杠需要写两个，下面会详细介绍。 十六进制数字可以大写也可以小写，并且数字对之间允许有空格（但不能是在一个数对， 也不能是在 `\x` 起始序列）。 十六进制格式能够与许多的外部应用程序和协议兼容，并且转换往往比逃逸快， 因此更倾向于这种用法。

例子：

```
SELECT E'\\xDEADBEEF';
```

8.4.2. `bytea` 逃逸格式

对于 `bytea` 格式来说，"escape"格式是一种传统的 PostgreSQL格式。它采用以ASCII字符序列来表示二进制串的方法， 同时将那些无法表示成ASCII字符的二进制串转换成特殊的逃逸序列。从应用的角度看， 如果代表字节的字符有意义，那么这种表示方法会很方便。但实际上， 这样做会模糊二进制字符串和字符字符串之间的区别，从而造成困扰， 同时筛选出的逃逸机制会显得很臃肿。因此对一些新应用应该恰当的避免这种格式。

当以逃逸格式录入 `bytea` 值时，你必须逃逸某些字节值， 同时可以逃逸所有字节值。通常，要逃逸一个字节值， 将它转换成反斜杠+三位八进制值的形式（或两个反斜杠，如果用逃逸字符串语法将值写成文本形式）。 另外，反斜杠本身（字节值92）可以用双反斜杠表示。Table 8-7 给出了必须逃逸的字符串，和替代的逃逸序列（如适用）。

Table 8-7. `bytea` 文本逃逸八进制

十进制数值	描述	输入逃逸形式	例子	输出形式
0	八进制的零	`E'\\000'`	`SELECT E'\\000'::bytea;`	`\000`
39	单引号	`''''` 或 `E'\\047'`	`SELECT E'\''::bytea;`	`'`
92	反斜杠	`E'\\\\'` 或 `E'\\134'`	`SELECT E'\\\\'::bytea;`	`\\`
0 to 31 and 127 to 255	"不可打印"八进制字符	`E'\\``_xxx'_` (八进制值)	`SELECT E'\\001'::bytea;`	`\001`

逃逸不可打印字节的要求因区域设置而异。在某些场合下， 你可以不逃逸它们。请注意Table 8-7 里的每个例子都是刚好一个字节长， 虽然输出形式比一个字符要长。

你必须写这么多反斜杠的原因，如Table 8-7所示， 是因为一个写成字符串文本的输入字符串必须通过PostgreSQL 服务器里的两个分析阶段。每一对反斜杠中的第一个会被字符串文本分析器理解成一个逃逸字符而消耗掉， 于是剩下的第二个反斜杠被 `bytea` 输入函数当作一个三位八进制值或者是逃逸另外一个反斜杠的开始。 比如，一个传递给服务器的字符串文本 `E'\\001'` 在通过字符串分析器之后会当作 `\001` 发送给 `bytea` 输入函数，在这里它被转换成一个十进制值为 1 的单个字节。请注意， 单引号字符(')不会被 `bytea` 特殊对待，它遵循字符串文本的普通规则。又见Section 4.1.2.1。

`Bytea` 字节也在输出中逃逸。通常，每个"不可打印" 的字节值都转化成对应的前导反斜杠的三位八进制数值。大多数"可打印的" 字节值是以客户端字符集的标准表现形式出现的。十进制值为 92(反斜杠)的字节在输出中双写。 细节在Table 8-8里描述。

Table 8-8. `bytea` 输出逃逸序列

字节的十进制值	描述	逃逸的输出形式	例子	输出结果
92	反斜杠	`\\`	`SELECT E'\\134'::bytea;`	`\\`
0 to 31 and 127 to 255	"不可打印"八进制字符	`\``_xxx_` (八进制值)	`SELECT E'\\001'::bytea;`	`\001`
32 to 126	"可打印"八进制字符	客户端字符集表现形式	`SELECT E'\\176'::bytea;`	`~`

根据你使用的前端不同，在是否逃逸 `bytea` 字符串的问题上你可能有一些额外的工作要做。比如，如果你的接口自动转换换行和回车，那你可能还要逃逸它们。

8.5. 日期/时间类型

Table 8-9显示了PostgreSQL 支持的SQL中所有日期和时间类型。 这些数据类型上可以进行的操作在Section 9.9中描述。 日期是按照公历计算的，甚至日历之前的年份也介绍了 （参阅Section B.4获取更多信息）。

Table 8-9. 日期/时间类型

名字	存储空间	描述	最低值	最高值	分辨率
`timestamp [(``_p_)] [without time zone]`	8字节	日期和时间(无时区)	4713 BC	294276 AD	1毫秒/14位
`timestamp [(``_p_)] with time zone`	8字节	日期和时间，有时区	4713 BC	294276 AD	1毫秒/14位
`date`	4字节	只用于日期	4713 BC	5874897 AD	1天
`time [(``_p_)] [without time zone]`	8字节	只用于一日内时间	00:00:00	24:00:00	1毫秒/14位
`time [(``_p_)] with time zone`	12字节	只用于一日内时间，带时区	00:00:00+1459	24:00:00-1459	1毫秒/14位
`interval [ _fields_ ] [ ( _p_ ) ]`	12字节	时间间隔	-178000000 年	178000000 年	1毫秒/14位

> **Note:** SQL标准要求仅仅将`timestamp`类型等于`timestamp without time zone`类型，PostgreSQL遵守这个行为。（7.3之前的版本将其看做`timestamp with time zone`。）`timestamptz`作为`timestamp with time zone`的缩写被接受；这是PostgreSQL 的一个扩展。

`time`，`timestamp`和`interval`接受一个可选的精度值 `_p_` 以指明秒域中小数部分的位数。没有明确的缺省精度， `_p_` 的范围对`timestamp`和`interval`类型是从0到6。

> **Note:** 如果`timestamp`数值是以8字节整数(目前的缺省)的方式存储的， 那么微秒的精度就可以在数值的全部范围内都可以获得。如果`timestamp` 数值是以双精度浮点数(一个废弃的编译时的选项)的方式存储的， 那么有效精度会小于 6 。 `timestamp`值是以 2000-01-01 午夜之前或之后的秒数存储的。 当`timestamp`值用浮点数实现时，微秒的精度是为那些在 2000-01-01 前后几年的日期实现的， 对于那些远一些的日子，精度会下降。注意，使用浮点时间允许一个比上面显示的更大的`timestamp`值变化范围：从4713BC到5874897AD。
>
> 同一个编译时选项也决定`time`和`interval`值是保存成浮点数还是八字节整数。 在以浮点数存储的时候，随着时间间隔的增加，大的`interval`数值的精度会降低。

对于`time`类型，如果使用了八字节的整数存储，那么`_p_` 允许的范围是从 0 到 6 ， 如果使用的是浮点数存储，那么这个范围是 0 到 10 。

`interval`类型有一个额外的选项，通过下下面词组之一限制存储的字段值：

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

注意如果同时指定了`_fields_`和`_p_`， `_fields_`必须包含`SECOND`，因为精度只应用于秒。

`time with time zone`类型是 SQL 标准定义的， 但是完整定义的有些方面会导致有问题的用法。在大多数情况下， `date`，`time`，`timestamp without time zone`，和`timestamp with time zone` 的组合就应该能提供一切应用需要的日期/时间的完整功能。

`abstime`和`reltime`类型是低分辨率类型，它们被用于系统内部。 我们反对你在应用中使用这些类型，因为这些内部类型可能会在未来的版本里消失。

8.5.1. 日期/时间输入

日期和时间的输入几乎可以是任何合理的格式，包括 ISO-8601 格式、SQL-兼容格式、 传统 POSTGRES格式、其它的形式。对于一些格式， 日期输入里的日、月、年可能会让人迷惑，因此系统支持自定义这些字段的顺序。 把DateStyle参数设置为 `MDY` 就按照"月-日-年"解析，设置为 `DMY` 就按照"日-月-年"解析，设置为 `YMD` 就按照"年-月-日"解析。

PostgreSQL在处理日期/时间输入上比SQL 标准要求的更灵活。参阅Appendix B 获取关于日期/时间输入的准确分析规则和可识别文本字段，包括月份、星期几、时区。

请记住任何日期或者时间的文本输入需要由单引号包围，就像一个文本字符串一样。 参考Section 4.1.2.7获取更多信息。 SQL要求使用下面的语法：

```
_type_ [ (_p_) ] '_value_'
```

可选的精度声明中的 `_p_` 是一个整数，表示在秒域中小数部分的位数， 我们可以对 `time` , `timestamp` , `interval` 类型声明精度。 允许的精度在上面已经说明。如果在常量声明中没有声明精度，缺省是文本值的精度。

8.5.1.1. 日期

Table 8-10显示了 `date` 类型可能的输入方式。

Table 8-10. Date Input

例子	描述
1999-01-08	ISO 8601格式(建议格式)，任何方式下都是 1999 年 1 月 8 号
January 8, 1999	在任何 `datestyle` 输入模式下都无歧义
1/8/1999	有歧义，在 `MDY` 下是一月八号；在 `DMY` 模式下是八月一日
1/18/1999	`MDY` 模式下是一月十八日，其它模式下被拒绝
01/02/03	`MDY` 模式下的 2003 年 1 月 2 日； `DMY` 模式下的 2003 年 2 月 1 日； `YMD` 模式下的 2001 年 2 月 3 日
1999-Jan-08	任何模式下都是 1 月 8 日
Jan-08-1999	任何模式下都是 1 月 8 日
08-Jan-1999	任何模式下都是 1 月 8 日
99-Jan-08	`YMD` 模式下是 1 月 8 日，否则错误
08-Jan-99	一月八日，除了在 `YMD` 模式下是错误的之外
Jan-08-99	一月八日，除了在 `YMD` 模式下是错误的之外
19990108	ISO 8601；任何模式下都是 1999 年 1 月 8 日
990108	ISO 8601；任何模式下都是 1999 年 1 月 8 日
1999.008	年和年里的第几天
J2451187	儒略日
January 8, 99 BC	公元前 99 年

8.5.1.2. 时间

当日时间类型是 `time [(``_p_ ` ) ] without time zone 和 `time [ (``_p_ ` ) ] with time zone。 只写 `time` 等效于 `time without time zone` 。

这些类型的有效输入由当日时间后面跟着可选的时区组成(参阅 Table 8-11和Table 8-12)。 如果在 `time without time zone` 类型的输入中声明了时区，那么它会被悄悄地忽略。 同样指定的日期也会被忽略，除非使用了一个包括夏令时规则的时区名，比如 `America/New_York` ，在这种情况下， 必须指定日期以确定这个时间是标准时间还是夏令时。时区偏移将记录在 `time with time zone` 中。

Table 8-11. 时间输入

例子	描述
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	与 04:05 一样；AM 不影响数值
`04:05 PM`	与 16:05 一样；输入小时数必须<= 12
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601
`04:05:06 PST`	缩写的时区
`2003-04-12 04:05:06 America/New_York`	用名字声明的时区

Table 8-12. 时区输入

例子	描述
`PST`	太平洋标准时间(Pacific Standard Time)
`America/New_York`	完整时区名称
`PST8PDT`	POSIX 风格的时区
`-8:00`	ISO-8601 与 PST 的偏移
`-800`	ISO-8601 与 PST 的偏移
`-8`	ISO-8601 与 PST 的偏移
`zulu`	军方对 UTC 的缩写(译注：可能是美军)
`z`	`zulu` 的缩写

参考Section 8.5.3以获取如何指定时区的更多信息。

8.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和时间的连接组成，后面跟着一个可选的时区，一个可选的 `AD` 或 `BC` 。另外， `AD` / `BC` 可以出现在时区前面， 但这个顺序并非最佳的。因此：

```
1999-01-08 04:05:06
```

和：

```
1999-01-08 04:05:06 -8:00
```

都是有效的数值，它是兼容ISO-8601 的。另外， 也支持下面这种使用广泛的格式：

```
January 8 04:05:06 1999 PST
```

SQL标准通过"+"或者"-"是否存在和时间后的失去偏移来区分 `timestamp without time zone` 和 `timestamp with time zone` 文本。因此， 根据标准，

```
TIMESTAMP '2004-10-19 10:23:54'
```

是一个 `timestamp without time zone` ， 而

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

是一个 `timestamp with time zone` 。PostgreSQL 从来不会在确定文本的类型之前检查文本内容，因此会把上面两个都看做是 `timestamp without time zone` 。因此要保证把上面的第二个当作 `timestamp with time zone` 看待，就要给它明确的类型：

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

如果一个文本已被确定是 `timestamp without time zone` ，PostgreSQL 将悄悄忽略任何文本中指出的时区。因此，生成的日期/时间值是从输入值的日期/时间字段衍生出来的， 并且没有就时区进行调整。

对于 `timestamp with time zone` ，内部存储的数值总是 UTC(全球统一时间， 以前也叫格林威治时间GMT)。如果一个输入值有明确的时区声明， 那么它将用该时区合适的偏移量转换成 UTC 。如果在输入字符串里没有时区声明， 那么它就假设是在系统的TimeZone参数里的那个时区， 然后使用这个 `timezone` 时区转换成 UTC 。

如果输出一个 `timestamp with time zone` ，那么它总是从 UTC 转换成当前的 `timezone` 时区，并且显示为该时区的本地时间。要看其它时区的该时间， 要么修改 `timezone` ，要么使用 `AT TIME ZONE` 构造 (参阅Section 9.9.3)。

在 `timestamp without time zone` 和 `timestamp with time zone` 之间的转换通常假设 `timestamp without time zone` 数值应该以 `timezone` 本地时间的形式接受或者写出。其它的时区可以用 `AT TIME ZONE` 的方式为转换声明。

8.5.1.4. 特殊值

PostgreSQL为方便起见支持在 Table 8-13里面显示的几个特殊输入值。值 `infinity` 和 `-infinity` 是特别在系统内部表示的， 并且将按照同样的方式显示；但是其它的都只是符号缩写， 在读取的时候将被转换成普通的日期/时间值。特别是 `now` 和相关的字符串在读取的时候就被转换成对应的数值。 所有这些值在 SQL 命令里当作普通常量对待时，都需要包围在单引号里面。

Table 8-13. 特殊日期/时间输入

输入字符串	适用类型	描述
`epoch`	`date` , `timestamp`	1970-01-01 00:00:00+00 (Unix 系统零时)
`infinity`	`date` , `timestamp`	比任何其它时间戳都晚
`-infinity`	`date` , `timestamp`	比任何其它时间戳都早
`now`	`date` , `time` , `timestamp`	当前事务的开始时间
`today`	`date` , `timestamp`	今日午夜
`tomorrow`	`date` , `timestamp`	明日午夜
`yesterday`	`date` , `timestamp`	昨日午夜
`allballs`	`time`	00:00:00.00 UTC

下列SQL兼容函数也可以用于获取对应数据类型的当前时间值： `CURRENT_DATE` , `CURRENT_TIME` , `CURRENT_TIMESTAMP` , `LOCALTIME` , `LOCALTIMESTAMP` 。后四个接受一个可选的精度声明(Section 9.9.4)。不过， 请注意这些 SQL 函数不是 被当作数据输入字符串识别的。

8.5.2. 日期/时间输出

日期/时间类型的输出格式可以设成 ISO 8601(默认)、SQL(Ingres)、 传统的 POSTGRES(Unix date格式)或German四种风格之一。 SQL标准要求使用 ISO 8601 格式。"SQL"输出格式的名字是历史偶然。 Table 8-14显示了每种输出风格的例子。 `date` 和 `time` 类型的输出当然只是给出的例子里面的日期和时间部分。

Table 8-14. 日期/时间输出风格

风格	描述	例子
`ISO`	ISO 8601，SQL 标准	`1997-12-17 07:37:16-08`
`SQL`	传统风格	`12/17/1997 07:37:16.00 PST`
`Postgres`	原始风格	`Wed Dec 17 07:37:16 1997 PST`
`German`	地区风格	`17.12.1997 07:37:16.00 PST`

> **Note:** ISO 8601指定用大写字母 `T` 分隔日期和时间。PostgreSQL 在输入上接受这种格式，但是在输出上使用一个空格而不是 `T`，就像上面显示的。 这样更易读而且和RFC 3399或者其他的数据库系统一致。

如果声明了 DMY 顺序，那么在SQL和 POSTGRES 风格里， 日期在月份之前出现，否则月份出现在日期之前(参阅Section 8.5.1 看看这个设置如何影响对输入值的解释)。Table 8-15 显示了一个例子。

Table 8-15. 日期顺序习惯

`datestyle` 设置	输入顺序	输出样例
`SQL, DMY`	`_日_ / _月_ / _年_`	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	`_月_ / _日_ / _年_`	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	`_日_ / _月_ / _年_`	`Wed 17 Dec 07:37:16 1997 PST`

用户可以用 `SET datestyle` 命令选取日期/时间的风格， 也可以在配置文件 `postgresql.conf` 中的DateStyle参数中设置，或者在服务器或客户端的 `PGDATESTYLE` 环境变量中设置。

我们也可以用格式化函数 `to_char` (参见Section 9.8) 来更灵活地控制时间/日期地输出。

8.5.3. 时区

时区和时区习惯不仅仅受地球几何形状的影响，还受到政治决定的影响。 到了 19 世纪，全球的时区变得稍微标准化了些，但是还是易于遭受随意的修改， 部分是因为夏时制规则。PostgreSQL 使用广泛使用的 `zoneinfo` （Olson）时区信息数据库有关历史时区的规则。 对于未来的时间，假设对于给定时区最近的规则将在未来继续无期限的遵守。

PostgreSQL在典型应用中尽可能与SQL 的定义相兼容。但SQL标准在日期/时间类型和功能上有一些奇怪的混淆。 两个显而易见的问题是：

- `date` 类型与时区没有联系，而 `time` 类型却有或可以有。 然而，现实世界的时区只有在与时间和日期都关联时才有意义， 因为时间偏移量(时差)可能因为实行类似夏时制这样的制度而在一年里有所变化。

- 缺省的时区用一个数字常量表示与UTC的偏移(时差)。因此， 当跨DST(夏时制)界限做日期/时间算术时， 我们根本不可能把夏时制这样的因素计算进去。

为了克服这些困难，我们建议在使用时区的时候，使用那些同时包含日期和时间的日期/时间类型。 我们建议不要使用 `time with time zone` 类型(尽管 PostgreSQL出于合理应用以及为了与SQL 标准兼容的考虑支持这个类型)。PostgreSQL 假设你用于任何类型的本地时区都只包含日期或时间(而不包含时区)。

在系统内部，所有日期和时间都用全球统一时间UTC格式存储， 时间在发给客户前端前由数据库服务器根据TimeZone 配置参数声明的时区转换成本地时间。

PostgreSQL允许你用三种方法指定时区：

- 完整的时区名。例如 `America/New_York` 。所有可以识别的时区名在 `pg_timezone_names` 视图中列出(参见 Section 47.71)。PostgreSQL 使用广泛使用的 `zoneinfo` 时区数据， 所以这些时区名在其它软件里也能被轻松的识别。
- 时区缩写。例如 `PST` 。这种缩写名通常只是定义了相对于 UTC 的偏移量， 而前一种完整的时区名可能还隐含着一组夏时制转换规则。 所有可以识别的时区缩写在 `pg_timezone_abbrevs` 视图中列出(参见 Section 47.70)。你不能设置 TimeZone 或log_timezone 配置参数为时区缩写，但是你可以在日期/时间输入值中结合 `AT TIME ZONE` 操作符使用时区缩写。
- 除完整的时区名及其缩写之外，PostgreSQL还接受 POSIX 风格的 `_STD_``_offset_` 或 `_STD_``_offset_``_DST_` 格式的时区，其中的 `_STD_` 是时区缩写、 `_offset_` 是一个相对于 UTC 的小时偏移量、 `_DST_` 是一个可选的夏时制时区缩写(假定相对于给定的偏移量提前一小时)。例如，如果 `EST5EDT` 不是一个已识别的时区名，那么它将等同于美国东部时间。如果存在夏时制时区名是当前时区名，根据 `zoneinfo` 时区数据库的 `posixrules` 条目中相同的夏时制事务规则，可以考虑使用这个特性。 在一个PostgreSQL标准安装中， `posixrules` 与 `US/Eastern` 相同，因此POSIX格式的时区声明遵循USA夏时制规则。如果需要，可以通过替换 `posixrules` 文件来调整该习惯。

简言之，这就是完整的时区名与时区缩写之间的差异：时区缩写总是代表一个相对于 UTC 的固定偏移量， 然而大多数完整的时区名隐含着一个本地夏令时规则，因此就有可能有两个相对于 UTC 的不同偏移量。

需要警惕的是，由于没有合理的时区缩写检查，POSIX格式的时区特点能导致静默的伪输入。 例如，使用 `SET TIMEZONE TO FOOBAR0` 时，实际上系统使用的是一个很特别的UTC缩写。另一个需要注意的是，在POSIX时区名中，积极的偏移用于*west*格林尼治位置。 在其他地方，PostgreSQL遵循ISO-8601规定， 即积极的时区偏移*east*格林威治。

总体而言，PostgreSQL 8.2 版本以后时区名在所有情况下都是大小写无关的。 而之前的版本在某些情况下是大小写敏感的。

无论是完整的时区名还是时区缩写都不是硬连接进服务器的，它们都是从安装目录下的 `.../share/timezone/` 和 `.../share/timezonesets/` 配置文件中获取的(参见Section B.3)。

可以在 `postgresql.conf` 文件里设置TimeZone 配置参数，或者用任何其它在Chapter 18描述的标准方法。 除此之外，还有好几种特殊方法可以设置它：

- 使用SQL命令 `SET TIME ZONE` 为会话设置时区， 这是 `SET TIMEZONE TO` 的一个可选的拼写方式，更加兼容标准。

- 如果在客户端设置了 `PGTZ` 环境变量，那么libpq 在连接时将使用这个环境变量给后端发送一个 `SET TIME ZONE` 命令。

8.5.4. 间隔输入

`interval` 类型值可以用下面的详细语法写：

```
[@] _quantity_ _unit_ [`_quantity_` `_unit_`...] [`_direction_`]
```

这里 `_quantity_` 是一个数字（可能已标记）； `_unit_` 可以是 `microsecond` ， `millisecond` ， `second` ， `minute` ， `hour` ， `day` ， `week` ， `month` ， `year` ， `decade` ， `century` ， `millennium` 或这些单位的缩写或复数。 `_direction_` 可以是 `ago` 或为空。 `@` 标记是可选的。不同的单位的数量被隐式地添加适当的计算符号。 `ago` 否定所有。如果IntervalStyle 设置为 `postgres_verbose` ，那么这个语法同样用于间隔输出。

可以在没有明确单位标记的情况下声明天，小时，分钟和秒。例如， `'1 12:59:10'` 等同于 `'1 day 12 hours 59 min 10 sec'` 。同样， 可以用一个破折号来声明一个年和月的组合，例如 `'200-10'` 等同于 `'200 years 10 months'` 。（事实上，SQL标准值允许短的格式， 并且当 `IntervalStyle` 设置为 `sql_standard` 时，用于输出）。

要么使用4.4.3.2的"format with designators"，要么使用4.4.3.3的 "alternative format"， 间隔值可以写为ISO 8601的时间间隔。格式如下：

```
P _quantity_ _unit_ [ `_quantity_` `_unit_` ...] [ T [ `_quantity_` `_unit_` ...]]
```

字符串必须以 `P` 开始，并且可以含有一个 `T` 用以指明一天中时间的格式。 可用单位的缩写在Table 8-16有说明。可以忽略单位， 也可以以任意顺序声明，但单位小于一天时必须在 `T` 之后。 尤其 `M` 的含义依赖于它在 `T` 之前或之后。

Table 8-16. ISO 8601 间隔单位的缩写

缩写	含义
Y	年
M	月（日期部分）
W	周
D	日
H	小时
M	分钟（时间部分）
S	秒

以缩写格式：

```
P [ `_years_`-`_months_`-`_days_` ] [ T `_hours_`:`_minutes_`:`_seconds_` ]
```

一个字符串必须以 `P` 开始，然后以 `T` 隔开日期和时间。 给出的值是如同ISO 8601日期的数字。

当用 `_fields_` 规范写一个时间间隔常数，或将一个字符串标记为用 `_fields_` 规范定义的一个间隔列时，未标记单位的解释由 `_fields_` 解释。如 `INTERVAL '1' YEAR` 读作1年，然而 `INTERVAL '1'` 代表1秒。 同样， `_fields_` 规范中"最低"有效字段值规定会被静默的忽略。如， `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` 会导致删除秒字段，而不是天字段。

根据SQL标准，间隔值的所有字段必须有相同的符号，因此前导负号可以用于所有字段；如 `'-1 2:03:04'` 中负号同时应用于天和小时/分钟/秒。PostgreSQL 允许字段有不同的标记，并且传统上，文本表述中的每个字段会被认为是独立标记的， 因此在这个例子中的小时/分钟/秒被认为是正值。如果 `IntervalStyle` 被设置为 `sql_standard` ，那么前导标记被认为是应用于所有字段的 （当然前提是没有再出现其他标记），否则会使用传统的PostgreSQL解释。为了避免这种歧义，如果任何字段是负的，建议为每个字段附上一个明确的标记。

PostgreSQL内部， `interval` 值被存储为月，日，秒的格式， 这是因为月中包含天数不同，并且如果进行了夏令时调整，那么一天可以有23或25小时。 当秒字段可以存储分数时，月和天字段可以是整数型。 由于时间间隔通常是由常量字符串或 `timestamp` 减法来定义的， 这种存储方法在大多数情况下很有效。 `justify_days` 和 `justify_hours` 函数可用于调整溢出正常范围值的天和小时。

在详细的输入格式，以及更紧凑的输入格式中，字段值可以有小数部分， 例如 `'1.5 week'` 或 `'01:02:03.45'` 。这种输入被转换成恰当的月， 天和秒来存储。由于这样会产生小数的月或天，因此在低阶字段中引入了分数， 使用1 month = 30 days 和 1 day = 24 hours的转换。例如， `'1.5 month'` 即1个月15天。输出中，只有秒可以写成分数形式。

Table 8-17中有一些有效的 `interval` 输入的例子。

Table 8-17. 间隔输入

示例	说明
1-2	SQL标准格式：一年两个月
3 4:05:06	SQL标准格式：3天4小时5分6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	传统Postgres格式: 1年2个月3天4小时5分钟6秒
P1Y2M3DT4H5M6S	ISO 8601 "带标识符格式":与上面相同含义
P0001-02-03T04:05:06	ISO 8601 "缩写格式":与上面相同含义

8.5.5. 间隔输出

间隔类型的输出格式可以用命令 `SET intervalstyle` 设置为下面四种类型：`sql_standard`，`postgres`，`postgres_verbose` 或 `iso_8601`。默认是 `postgres` 格式，Table 8-18中有每种格式的示例。

`sql_standard` 格式产生的输出结果符合SQL的间隔字符串标准， 如果间隔值满足标准的限制（无论只有年-月，或只有天-时间，没有积极和消极的构成的混合）。 否则输出类似一个标准年-月文本字符串后跟有一个天-时间文本字符串， 带有添加明确标记的消除歧义混合信号的时间间隔。

当参数DateStyle设置为 `ISO` 时， `postgres` 格式的输出与PostgreSQL 8.4之前的版本一致。

当参数 `DateStyle` 设置为非- `ISO` ， `postgres_verbose` 格式的输出与PostgreSQL 8.4之前的版本一致。

`iso_8601` 格式的输出与ISO 8601标准4.4.3.2节中的"format with designators"一致。

Table 8-18. 间隔输出格式示例

格式	年-月间隔	天-时间间隔	混合间隔
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 年 2 个月	3 天 04:05:06	-1 年 -2 个月 +3 天 -04:05:06
`postgres_verbose`	@ 1 年 2 个月	@ 3 天 4 小时 5 分 6 秒	@ 1 年 2 个月 -3 天 4 小时 5 分 6 秒以前
`iso_8601`	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. 布尔类型

PostgreSQL支持SQL标准的 `boolean` 数据类型。参阅Table 8-19。 `boolean` 只能有"true"(真)或"false"(假)两个状态， 或第三种"unknown"(未知)状态，用 NULL 表示。

Table 8-19. 布尔数据类型

名称	存储格式	描述
`boolean`	1 字节	真/假

"真"值的有效文本值是：

TRUE
`'t'`
`'true'`
`'y'`
`'yes'`
`'on'`
`'1'`

对于"假"，你可以使用下面这些：

FALSE
`'f'`
`'false'`
`'n'`
`'no'`
`'off'`
`'0'`

前导或尾随空白将被忽略，大小写无关。使用 `TRUE` 和 `FALSE` 这样的字眼比较好(也是SQL兼容的用法)。

Example 8-2显示了用字母 `t` 和 `f` 输出 `boolean` 值的例子。

Example 8-2. 使用 `boolean` 类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

8.7. 枚举类型

枚举类型是一个包含静态和值的有序集合的数据类型。等于某些编程语言中的 `enum` 类型。一个枚举类型可以是一周中的天，或者一块数据的状态值的集合。

8.7.1. 枚举类型的声明

用CREATE TYPE创建枚举类型，如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

就像其他类型一样，一旦创建，枚举类型可以用于表和函数定义。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. 排序

枚举类型中，值的顺序是创建枚举类型时定义的顺序。 所有的比较标准运算符及其相关的聚集函数都可支持枚举类型，例如：

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. 类型安全

每个枚举类型都是独立的，不能与其他枚举类型结合，如：

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果真的需要那么做，可以要么自定义运算符，要么为查询添加显式转换：

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. 实施细则

一个枚举值在磁盘上占4字节。一个枚举值的文本标签长度由编译到PostgreSQL 中的 `NAMEDATALEN` 设置，以标准方式编译意味着最多63字节。

枚举标签对大小写是敏感的，因此 `'happy'` 不等于 `'HAPPY'` 。 标签中的空格也是一样。

从内部枚举值到文本标签的翻译是保存在系统目录 `pg_enum` 中。 可以直接查询这个目录。

8.8. 几何类型

几何数据类型表示二维的平面物体。Table 8-20 显示了PostgreSQL里面可用的几何类型。 最基本的类型：点，是其它类型的基础。

Table 8-20. 几何类型

名字	存储空间	说明	表现形式
`point`	16 字节	平面中的点	(x,y)
`line`	32 字节	(无穷)直线(未完全实现)	((x1,y1),(x2,y2))
`lseg`	32 字节	(有限)线段	((x1,y1),(x2,y2))
`box`	32 字节	矩形	((x1,y1),(x2,y2))
`path`	16+16n 字节	闭合路径(与多边形类似)	((x1,y1),...)
`path`	16+16n 字节	开放路径	[(x1,y1),...]
`polygon`	40+16n 字节	多边形(与闭合路径相似)	((x1,y1),...)
`circle`	24 字节	圆	<(x,y),r> (圆心和半径)

我们有一系列丰富的函数和操作符可用来进行各种几何计算，如拉伸、转换、旋转、计算相交等。 它们在Section 9.11里有解释。

8.8.1. 点

点是几何类型的基本二维构造单位。用下面语法描述 `point` 的数值：

```
( _x_ , _y_ )
  _x_ , _y_
```

这里的 `_x_` 和 `_y_` 是用浮点数表示的点的坐标。

点输出使用第一种语法。

8.8.2. 线段

线段(`lseg`)是用一对点来代表的。 `lseg` 的值用下面语法声明：

```
[ ( _x1_ , _y1_ ) , ( _x2_ , _y2_ ) ]
( ( _x1_ , _y1_ ) , ( _x2_ , _y2_ ) )
  ( _x1_ , _y1_ ) , ( _x2_ , _y2_ )
    _x1_ , _y1_   ,   _x2_ , _y2_
```

这里的 `(``_x1_` , `_y1_` ) 和 `(``_x2_` , `_y2_`)是线段的端点。

线段输出使用第一种语法。

8.8.3. 矩形

矩形是用一对对角点来表示的。 `box` 的值用下面语法声明：

```
( ( _x1_ , _y1_ ) , ( _x2_ , _y2_ ) )
  ( _x1_ , _y1_ ) , ( _x2_ , _y2_ )
    _x1_ , _y1_   ,   _x2_ , _y2_
```

这里的 `(``_x1_` , `_y1_` ) 和 `(``_x2_` , `_y2_`) 是矩形的一对对角点。

矩形的输出使用第二种语法。

任何两个对角都可以出现在输入中，但按照那样的顺序， 右上角和左下角的值会被重新排序以存储。

8.8.4. 路径

路径由一系列连接的点组成。路径可能是开放的， 也就是认为列表中第一个点和最后一个点没有连接，也可能是闭合的， 这时认为第一个和最后一个点连接起来。

`path` 的数值用下面语法声明：

```
[ ( _x1_ , _y1_ ) , ... , ( _xn_ , _yn_ ) ]
( ( _x1_ , _y1_ ) , ... , ( _xn_ , _yn_ ) )
  ( _x1_ , _y1_ ) , ... , ( _xn_ , _yn_ )
  ( _x1_ , _y1_   , ... ,   _xn_ , _yn_ )
    _x1_ , _y1_   , ... ,   _xn_ , _yn_
```

这里的点是组成路径的线段的端点。方括弧(`[]`)表明一个开放的路径， 圆括弧(`()`)表明一个闭合的路径。当最外层的括号被省略， 如在第三至第五语法，会假定一个封闭的路径。

路径的输出使用第一种或第二种语法输出，在适当的时候。

8.8.5. 多边形

多边形由一系列点代表(多边形的顶点)。多边形可以认为与闭合路径一样，但是存储方式不一样而且有自己的一套支持函数。

`polygon` 的数值用下列语法声明：

```
( ( _x1_ , _y1_ ) , ... , ( _xn_ , _yn_ ) )
  ( _x1_ , _y1_ ) , ... , ( _xn_ , _yn_ )
  ( _x1_ , _y1_   , ... ,   _xn_ , _yn_ )
    _x1_ , _y1_   , ... ,   _xn_ , _yn_
```

这里的点是多边形的端点。

多边形输出使用第一种语法。

8.8.6. 圆

圆由一个圆心和半径标识。 `circle` 的数值用下面语法表示：

```
< ( _x_ , _y_ ) , _r_ >
( ( _x_ , _y_ ) , _r_ )
  ( _x_ , _y_ ) , _r_
    _x_ , _y_   , _r_
```

这里的 `(``_x_` , `_y_` ) 是圆心， `_r_` 是半径。

圆的输出用第一种格式。

8.9. 网络地址类型

PostgreSQL提供用于存储 IPv4 、IPv6 、MAC 地址的数据类型， 在Table 8-21里显示。 用这些数据类型存储网络地址比用纯文本类型好， 因为这些类型提供输入错误检查和特殊的操作和功能(见Section 9.12)。

Table 8-21. 网络地址类型

名字	存储空间	描述
`cidr`	7 或 19 字节	IPv4 或 IPv6 网络
`inet`	7 或 19 字节	IPv4 或 IPv6 主机和网络
`macaddr`	6 字节	MAC 地址

在对 `inet` 或 `cidr` 数据类型进行排序的时候， IPv4 地址总是排在 IPv6 地址前面，包括那些封装或者是映射在 IPv6 地址里的 IPv4 地址， 比如 ::10.2.3.4 或 ::ffff:10.4.3.2。

8.9.1. `inet`

`inet` 在一个数据域里保存主机的 IPv4 或 IPv6 地址，以及一个可选的等效子网。 子网的等效是通过计算主机地址中有多少位表示网络地址的方法来表示的("子网掩码")。 如果子网掩码是 32 并且地址是 IPv4 ，那么不表示任何子网，只是一台主机。 在 IPv6 里，地址长度是 128 位，因此 128 位表明一个唯一的主机地址。 请注意如果你想只接受网络地址，你应该使用 `cidr` 类型而不是 `inet` 类型。

该类型的输入格式是 `_address/y_` ， 这里的 `_address_` 是 IPv4 或者 IPv6 地址， `_y_` 是子网掩码的二进制位数。如果 `_/y_` 部分未填，则子网掩码对 IPv4 而言是 32 ， 对 IPv6 而言是 128 ， 所以该值表示只有一台主机。显示时，如果该值表示只有一台主机， `_/y_` 将不会显示。

8.9.2. `cidr`

`cidr` 保存一个 IPv4 或 IPv6 网络地址。其输入和输出遵循无类别的互联网域路由习惯。 声明一个网络的格式是 `_address/y_` ， 这里的 `_address_` 是 IPv4 或者 IPv6 地址， `_y_` 是子网掩码的二进制位数。如果省略 `_y_` ， 那么掩码部分用旧的有类别的网络编号系统进行计算，但要求输入的数据已经包括了确定掩码所需的所有字节。 声明一个指定掩码的网络地址是错误的。

Table 8-22 是些例子。

Table 8-22. `cidr` 类型输入举例

`cidr` 输入	`cidr` 输出	
192.168.100.128/25	192.168.100.128/25	192.168.
192.168/24	192.168.0.0/24	192.168.
192.168/25	192.168.0.0/25	192.168.
192.168.1	192.168.1.0/24	192.168.
192.168	192.168.0.0/24	192.168.
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/2
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3

8.9.3. `inet` 对比 `cidr`

`inet` 和 `cidr` 类型之间的基本区别是 `inet` 接受子网掩码，而 `cidr` 不接受。

> **Tip:** 如果你不喜欢 `inet` 或 `cidr` 值的输出格式， 请试一下 `host`，`text` 和 `abbrev` 函数。

8.9.4. `macaddr`

`macaddr` 类型存储 MAC 地址，也就是以太网卡硬件地址(尽管 MAC 地址还用于其它用途)。可以接受下列格式：

'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'

它们声明的都是同一个地址。对于数据位 `a` 到 `f`， 大小写都行。输出总是我们上面给出的第一种形式。

IEEE标准802-2001指定第二种形式（带连字符）为MAC地址的标准格式， 并指定的第一种形式（用冒号）为位反转符号，因此08-00-2b-01-02-03=01:00:4D:08:04:0C。 这个条约现在已很少使用，它和过时的网络协议（如令牌环）有关。 PostgreSQL对位反转没有规定，并且所有接受的格式使用LSB协议顺序。

其余四个输入格式不是任何标准的一部分。

8.10. 位串类型

位串就是一串 1 和 0 的字符串。它们可以用于存储和直观化位掩码。 我们有两种 SQL 位类型： `bit(``_n_ )` 和 `bit varying(``_n_)`， 这里的 `_n_` 是一个正整数。

`bit` 类型的数据必须准确匹配长度 `_n_` ， 试图存储短些或者长一些的数据都是错误的。 `bit varying` 类型数据是最长 `_n_` 的变长类型；更长的串会被拒绝。 写一个没有长度的 `bit` 等效于 `bit(1)` ， 没有长度的 `bit varying` 意思是没有长度限制。

> **Note:** 如果我们明确地把一个位串值转换成 `bit(``_n_ )`， 那么它的右边将被截断或者在右边补齐零，直到刚好 `_n_` 位， 而不会抛出任何错误。类似地，如果我们明确地把一个位串数值转换成 `bit varying(``_n_)`，如果它超过了 `_n_` 位， 那么它的右边将被截断。

请参考Section 4.1.2.5获取有关位串常量的语法信息。 还有一些位逻辑操作符和位处理函数可用；参见Section 9.6。

Example 8-3. 使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
<samp class="literal">ERROR:  bit string length 2 does not match type bit(3)</samp>
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
 <samp class="literal">a  |  b
-----+-----
 101 | 00
 100 | 101</samp>
```

位字符串值需要1字节，每组8位，增加5或8字节的开销取决于字符串的长度 （但是长值会被压缩或移动到另外一条线上， Section 8.3中有相关字符串的解释）。

8.11. 文本搜索类型

PostgreSQL提供了两种数据类型用于支持全文检索， 即通过自然语言*documents*的集合来找到那些匹配一个*query* 的检索。 `tsvector` 类型产生一个文档（以优化了全文检索的形式）， `tsquery` 类型用于代表查询。Chapter 12 中说明了这两个类型，同时Section 9.13 总结了相关的函数和操作符。

8.11.1. `tsvector`

`tsvector` 的值是一个无重复值的*lexemes*排序列表， 即一些同一个词的不同变种的标准化（可参阅Chapter 12）。 在输入的同时会自动排序和消除重复，如：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

为了包含空格或标点符号，可以用引号标记：

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

（在这个例子中，我们使用了双引号美元字符串文本， 下一个例子是为了避免文本中双引号的混淆。）枚举的引号和反斜杠必须双倍：

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

可选的，整型*positions*也可以放到词汇中：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

位置通常表示文档中的源字的位置。位置信息可以用于*proximity ranking*。 位置值的范围是1到16383，最大值默认是16383。相同词的重复位会被忽略掉。

拥有位置的词汇甚至可以用一个权来标记，这个权可以是 `A`， `B`， `C` 或 `D`。 默认的是 `D`，因此输出中不会出现：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

权可以用来反映文档结构，如：标记标题以与主体相区别。 全文检索排序函数可以为不同的权标记来分配不同的优先级。

充分理解 `tsvector` 类型不能自己标准化这一点是很重要的， 它假设传递给它的单词对应用程序来说是恰当的标准化了的，如：

```
select 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

对大多数的英文全文检索应用来说，上面的单词会被认为非规范化的， 但 `tsvector` 并不关心这些。原始文件中的文字应该通过 `to_tsvector` 来为检索恰当的规范化这些单词:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

详细信息可参阅Chapter 12。

8.11.2. `tsquery`

`tsquery` 存储用于检索的词汇，并且使用布尔操作符 `&` (AND)， `|` (OR)和 `!` (NOT) 来组合它们。括号用来强调操作符的分组：

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

在没有括号的情况下， `!` (NOT)结合的最紧密， 而 `&` (AND)结合的比 `|` (OR)紧密。

可选的， `tsquery` 中的词汇可以被一个或多个权字母来标记， 这些权字母用来限制它们只能与带有匹配权的 `tsvector` 词汇进行匹配。

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

同样， `tsquery` 中的词汇可以用 `*` 进行标记来指定前缀匹配：

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

这个查询可以匹配 `tsvector` 中以"super"开始的任意单词。 请注意，前缀首先被文本搜索配置处理，这也就意味着下面的结果为真：

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
(1 row)
```

因为 `postgres` 去掉后面后得到 `postgr` ：

```
SELECT to_tsquery('postgres:*');
 to_tsquery
------------
 'postgr':*
(1 row)
```

这样就匹配 `postgraduate` 了。

词汇的引用规则与之前 `tsvector` 中词汇的描述一样；并且， 与 `tsvector` 一样，任何单词必须在转换为 `tsquery` 类型前规范化。 `to_tsquery` 函数可以方便的用来执行规范化。

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

8.12. UUID 类型

`uuid` 数据类型用来存储RFC 4122，ISO/IEF 9834-8:2005以及相关标准定义的通用唯一标识符（UUID）。 （一些系统认为这个数据类型为全球唯一标识符，或GUID。） 这个标识符是一个由算法产生的128位标识符，使它不可能在已知使用相同算法的模块中和其他方式产生的标识符相同。 因此，对分布式系统而言，这种标识符比序列能更好的提供唯一性保证，因为序列只能在单一数据库中保证唯一。

UUID被写成一个小写十六进制数字的序列，由分字符分成几组， 特别是一组8位数字+3组4位数字+一组12位数字，总共32个数字代表128位， 一个这种标准的UUID例子如下：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL同样支持以其他方式输入：大写数字， 由花括号包围的标准格式，省略部分或所有连字符，在任意一组四位数字之后加一个连字符。如：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

一般是以标准格式输出。

PostgreSQL为UUID提供了存储和比较函数， 但核心数据库不包括能生成UUID的函数，因为没有单一的算法非常适合于每一个应用程序。 uuid-ossp模块提供了实施几个标准算法的函数。另外， UUID可以由客户端应用或通过服务器端函数库调用而生成。

8.13. XML 类型

`xml` 数据类型可以用于存储XML数据。 将XML数据存到 `text` 类型中的优势在于它能够为结构良好性来检查输入值， 并且还支持函数对其进行类型安全性检查，可参阅Section 9.14。 要使用这个数据类型，编译时必须使用 `configure --with-libxml` 。

`xml` 可以存储由XML标准定义的格式良好的"文档"， 以及由XML标准中的 `XMLDecl? content` 定义的"内容"片段， 大致上，这意味着内容片段可以有多个顶级元素或字符节点。 `_xmlvalue_` IS DOCUMENT 表达式可以用来判断一个特定的 `xml` 值是一个完整的文件还是内容片段。

8.13.1. 创建XML值

使用函数 `xmlparse` : 来从字符数据产生 `xml` 类型的值 :

```
XMLPARSE ( { DOCUMENT | CONTENT } _value_)
```

例子 :

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

然而根据SQL标准，这是唯一的用于将字符串转换成XML值的方式，PostgreSQL特有的语法也可以使用 :

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

`xml` 类型对一个文档类型声明（DTD）不会验证输入值， 即使输入值声明了一个DTD。目前没有内置支持用于对其他XML架构语言（如XML Schema）验证。

使用函数 `xmlserialize` : 来从 `xml` 产生一个字符串。

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } _value_ AS _type_ )
```

`_type_` 可以是 `character` ， `character varying` 或 `text` (或其中某个的变种)。同时，根据SQL标准， 这是 `xml` 和字符类型之间的唯一的转换方式，但PostgreSQL仍支持简单的值转换。

当一个字符串值在没有通过 `XMLPARSE` 或 `XMLSERIALIZE` 的情况下， 与 `xml` 类型进行转换时，分别的，选择 `DOCUMENT` 与 `CONTENT` 是由"XML option" 会话配置参数决定，这个配置参数可以由标准命令来设置：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

或更多类似的PostgreSQL语法：

```
SET xmloption TO { DOCUMENT | CONTENT };
```

默认是 `CONTENT` ，因此所有的XML数据格式都能支持。

> **Note:** 随着默认XML选项的设置，如果字符串中包含一个文档类型声明， 那么你不能直接将其转换成 `xml` 类型，因为XML内容片断的定义不支持。 如果非得需要这么做，要么使用 `XMLPARSE` ，要么更改XML选项。

8.13.2. 编码处理

在对客户端和服务器端进行多字符编码，以及在通过它们传递XML数据时需要格外注意。 当使用文本模式（正常模式）在服务器端和客户端之间传递查询和查询结果时， PostgreSQL在各自终端对所有传递的字符数据和字符编码进行相互转换，参阅Section 22.3。 这包括XML值的字符串表示形式，如上面的例子。这通常意味着XML数据中的编码声明， 在客户端和服务器之间传递时，可以成为无效字符数据转换为其他编码。 这是因为枚举编码声明没有改变。为了应对该问题， 提交输入到 `xml` 类型的字符串中的编码声明会被*ignored*， 同时，内容会被认为是在当前服务器编码中。所以，对正确的处理来说， XML数据的字符串必须从在当前客户端编码中的客户端发送。客户端有责任， 要么在传递到服务器之前将文档转换成当前客户端编码，要么适当的调整客户端编码。 输出时， `xml` 类型的值不会有编码声明， 同时客户端会认为所有的数据都是在当前客户端编码之中的。

当使用二进制模式在服务器和客户端之间传递查询参数和查询结果，没有执行字符集转换，因此解决方法是不同的。在这种情况下，将会遵守XML数据中的编码声明， 并且如果声明不存在，数据会被假定为UTF-8格式(如同XML标准要求那样， 但需要注意的是PostgreSQL不支持UTF-16)。输出时，会对数据进行编码声明以指定客户端编码， 除非客户端编码格式是UTF-8。

不用说，如果XML数据编码格式，客户端编码格式，以及服务器编码格式都一样， 那么用PostgreSQL处理XML数据将会减少错误，并且效率会很高。在内部， XML数据是用UTF-8编码格式处理的，因此，如果服务器端编码也是UTF-8时，计算性能会很高。

Caution

当服务器编码非UTF-8格式时，一些XML相关的函数可能完全不支持非ASCII数据， 特别是`xpath()`函数。

8.13.3. 访问XML值

`xml`数据类型有些特殊，因为它不提供比较运算符。这是因为对XML数据， 没有很好的定义和通用的比较运算符。这样做的一个后果是， 不能通过`xml`与检索值的比较来检索行。因此XML值必须带有一个单独的关键值， 如一个ID。另一个解决比较XML值的方法是，先将它们转换成字符串， 但需要注意的是字符串比较与一个有用的XML比较方法无关。

因为没有针对`xml`数据类型的比较运算符，因此不能在这种类型的字段上直接创建索引。 如果需要对XML数据进行快速搜索，可能的解决方法包括将表达式转换成一个字符串类型， 然后对它进行索引，或索引一个XPath表达式。当然，实际查询是不得不进行调整， 以使用一个索引表达式进行检索。

PostgreSQL中的文本检索功能也可用于加快XML数据的全文搜索。 但必要的预处理支持在PostgreSQL中还不能获得。

8.14. JSON 类型

`json` 数据类型可以用来存储JSON（JavaScript Object Notation）数据， 就像RFC 4627中指定的那样。 这样的数据也可以存储为 `text` ，但是 `json` 数据类型更有利于检查每个存储的数值是可用的JSON值。 这里也有相关的支持函数可用；参阅Section 9.15。

PostgreSQL只允许每个数据库用一种服务器编码。因此， 使JSON严格符合规范是不可能的，除非服务器编码是UTF-8。 试图直接包含不能在服务器编码中表示的字符将会失败；相反的， 能在服务器编码中表示但是不在UTF-8中的字符是被允许的。 `\uXXXX` 逃逸是被允许的，无视服务器编码，并且只检查语法的正确性。

8.15. Arrays

PostgreSQL允许将字段定义成变长的多维数组。 数组类型可以是任何基本类型或用户定义类型，枚举类型或复合类型。 不支持域的数组。

8.15.1. 数组类型的声明

为说明这些用法，我们先创建一个由基本类型数组构成的表：

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

如上所示，一个数组类型是通过在数组元素类型名后面附加方括弧(`[]`)来命名的。 上面的命令将创建一个叫 `sal_emp` 的表，表示雇员名字的 `name` 字段是一个 `text` 类型字符串，表示雇员季度薪水的 `pay_by_quarter` 字段是一个一维 `integer` 数组，表示雇员周计划的 `schedule` 字段是一个两维 `text` 数组。

`CREATE TABLE` 的语法允许声明数组的确切大小，比如：

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

不过，目前的实现忽略任何提供的数组尺寸限制(等价于未声明长度的数组)。

目前的实现也不强制数组维数。特定元素类型的数组都被认为是相同的类型， 不管他们的大小或者维数。因此，在 `CREATE TABLE` 里定义数字或者维数都不影响运行时的行为。

另外还有一种语法，它通过使用关键字 `ARRAY` 遵循 SQL 标准， 可以用于声明一维数组。 `pay_by_quarter` 可以定义为：

```
    pay_by_quarter  integer ARRAY[4],
```

或者不声明数组的大小：

```
    pay_by_quarter  integer ARRAY,
```

不过，如前所述，PostgreSQL并不强制这个尺寸限制。

8.15.2. 数组值输入

将数组写成文本的时候，用花括弧把数组元素括起来并且用逗号将它们分开(如果你懂 C， 那么这与初始化一个结构很像)。你可以在数组元素值周围放置双引号， 但如果这个值包含逗号或者花括弧，那么就必须加上双引号(下面有更多细节)。 因此，一个数组常量的常见格式如下：

```
'{ _val1_ _delim_ _val2_ _delim_ ... }'
```

这里的 `_delim_` 是该类型的分隔符，就是在该类型的 `pg_type` 记录中指定的那个。在 PostgreSQL 发布提供的标准数据类型里，所有类型都使用逗号(`,`),除了 `box` 类型使用分号(`;`)之外。每个 `_val_` 要么是一个数组元素类型的常量，要么是一个子数组。一个数组常量的例子如下：

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

这个常量是一个 3 乘 3 的两维数组，由三个整数子数组组成。

要将一个数组元素的值设为 NULL， 直接写上 `NULL` 即可(大小写无关)。 要将一个数组元素的值设为字符串"NULL"，那么你必须加上双引号。

这种数组常量实际上只是我们在Section 4.1.2.7 里讨论过的一般类型常量的一种特例。常量最初是当作字符串看待并且传递给数组输入转换器的， 可能需要使用明确的类型声明。

现在我们可以展示一些 `INSERT` 语句。

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

前面的两个插入的结果看起来像这样：

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                 schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多维数组必须匹配每个维的元素数。如果不匹配将导致错误：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

我们还可以使用 `ARRAY` 构造器语法：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

请注意数组元素是普通的 SQL 常量或者表达式；比如，字符串文本是用单引号包围的， 而不是像数组文本那样用双引号。 `ARRAY` 构造器语法在Section 4.2.12 里有更详细的讨论。

8.15.3. 访问数组

现在我们可以在这个表上运行一些查询。首先，我们演示如何访问数组的一个元素。 这个查询检索在第二季度薪水变化的雇员名：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

数组的下标数字是写在方括弧内的。PostgreSQL 缺省使用以 1 为基的数组习惯，也就是说，一个 `_n_` 元素的数组从 `array[1]` 开始，到 `array[``_n_`]结束。

这个查询检索所有雇员第三季度的薪水：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

我们还可以访问一个数组的任意矩形片段，或称子数组。对于一维或更多维数组， 可以用 `_下标下界_`:`_下标上界_` 表示一个数组的某个片段。比如，下面查询检索 Bill 该周头两天的第一件计划：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果任意维数被写为一个片段，也就是，包含一个冒号，那么所有维数都被当做是片段。 任意只有一个数字（没有冒号）的维数是从1开始到声明的数字为止的。例如， `[2]` 被认为是 `[1:2]` ，就想下面例子中一样：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                 schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

为了与没有片段的情况相区分，最好是对所有维数都使用片段语法，例如， `[1:2][1:1]` ， 而不是 `[2][1:1]` 。

如果数组本身或任何下标表达式是 NULL ，那么该数组的下标表达式也将生成 NULL 。 从一个数组的当前范围之外抓取数据将生成一个 NULL ， 而不是导致错误。 比如， 如果 `schedule` 目前的维是 `[1:3][1:2]` ， 然后我们抓取 `schedule[3][3]` 会生成 NULL 。类似的还有， 一个下标错误的数组引用也生成 NULL ， 而不是错误。

如果数组本身或任何下标表达式是 NULL ，那么该数组的片段表达式也将生成 NULL 。 但在其它其它情况下，比如抓取一个完全在数组的当前范围之外的数组片断， 将生成一个空数组(零维)而不是 NULL 。（这不匹配无片段数组的行为并且是为历史原因这样做的。） 如果抓取的片断部分覆盖数组的范围，那么它会自动缩减为抓取覆盖的范围而不是返回null。

任何数组的当前维数都可以用 `array_dims` 函数检索：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` 生成一个 `text` 结果， 对于人类可能比较容易阅读，但是对于程序可能就不那么方便了。 我们也可以用 `array_upper` 和 `array_lower` 函数分别返回数组特定维的上界和下界:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` 将返回特定维数数组的长度：

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

8.15.4. 修改数组

一个数组值可以完全被代替：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

或者使用 `ARRAY` 构造器语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

或者只是更新某一个元素：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

或者更新某个片断：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

可以通过给一个尚不存在数组元素赋值的办法扩大数组， 所有位于原数组最后一个元素和这个新元素之间的未赋值元素都将设为 NULL 。 例如，如果 `myarray` 数组当前有 4 个元素， 在对 `myarray[6]` 赋值之后它将拥有 6 个元素，其中 `myarray[5]` 的值将为 NULL 。目前，只允许对一维数组使用这种方法扩大(对多维数组行不通)。

下标赋值允许创建下标不从 1 开始的数组。比如，我们可以给 `myarray[-2:7]` 赋值，创建一个下标值在 -2 到 7 之间的数组。

新的数组值也可以用连接操作符 `||` 构造:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

连接操作符允许把一个元素压入一维数组的开头或者结尾。它还接受两个 `_N_` 维的数组，或者一个 `_N_` 维和一个 `_N+1_` 维的数组。

当向一维数组的头部或尾部压入单独一个元素后，数组的下标下界保持不变。比如：

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

如果将两个相同维数的数组连接在一起，结果数组将保持左操作数的外层维数的下标下界。结果是这样一个数组：包含左操作数的每个元素，后面跟着右操作数的每个元素。比如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

如果将一个 `_N_` 维的数组压到一个 `_N+1_` 维数组的开头或者结尾， 结果和上面数组元素的情况类似。每个 `_N_` 维的子数组实际上都是 `_N+1_` 维数组的最外层的元素。比如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

数组也可以用 `array_prepend`，`array_append`，`array_cat` 函数构造。前两个只支持一维数组，而 `array_cat` 支持多维数组。请注意使用上面讨论的连接操作符要比直接使用这些函数好。实际上， 这些函数主要用于实现连接操作符。不过，在用户定义的创建函数里直接使用他们可能有必要。 一些例子：

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

8.15.5. 在数组中检索

要搜索一个数组中的数值，你必须检查该数组的每一个值。你可以手工处理(如果你知道数组尺寸)。比如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

不过，对于大数组而言，这个方法很快就会让人觉得无聊，并且如果你不知道数组尺寸， 那就没什么用了。另外一个方法在Section 9.23 里描述。上面的查询可以用下面的代替：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

另外，你可以用下面的语句找出数组中所有元素值都等于 10000 的行：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

或者，可以使用`generate_subscripts`函数。例如：

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

这个函数在Table 9-50里面描述。

你可以使用`&&`操作符检索一个数组，它可以检查左操作数是否与右操作数重叠。 例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

这个操作符和另外一个数组操作符在Section 9.18里有详细的描述。 它可以通过一个恰当的索引加速，在Section 11.2里面描述。

> **Tip:** 数组不是集合；需要像前面那样搜索数组中的特定元素通常表明你的数据库设计有问题。 数组字段通常是可以分裂成独立的表。很明显表要容易搜索得多， 并且在元素数目非常庞大的时候也可以更好地伸展。

8.15.6. 数组输入和输出语法

一个数组值的外部表现形式由一些根据该数组元素类型的 I/O 转换规则分析的项组成， 再加上一些标明该数组结构的修饰。这些修饰由围绕在数组值周围的花括弧(`{` 和 `}`)加上相邻项之间的分隔字符组成。分隔字符通常是一个逗号(`,`) 但也可以是其它的东西：它由该数组元素类型的`typdelim`设置决定。 在PostgreSQL提供的标准数据类型里，所有类型都使用逗号， 除了`box`类型使用分号(`;`)外。在多维数组里， 每个维都有自己级别的花括弧，并且在同级相邻的花括弧项之间必须写上分隔符。

如果数组元素值是空字符串或者包含花括弧、分隔符、双引号、反斜杠、空白， 或者匹配关键字`NULL`，那么数组输出过程将在这些值周围包围双引号。 在元素值里包含的双引号和反斜杠将被反斜杠逃逸。对于数值数据类型， 你可以安全地假设数值没有双引号包围，但是对于文本类型， 我们就需要准备好面对有双引号包围和没有双引号包围两种情况了。

缺省时，一个数组的某维的下标索引是设置为 1 的。如果一个数组的某维的下标不等于 1 ，那么就会在数组结构修饰域里面放置一个实际的维数。 这个修饰由方括弧(`[]`)围绕在每个数组维的下界和上界索引， 中间有一个冒号(`:`)分隔的字符串组成。 数组维数修饰后面跟着一个等号操作符(`=`)。比如：

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

仅当一个或多个下界不等于 1 时，数组输出程序才在结果中包含明确的尺寸。

如果一个数组元素的值写成 `NULL` (无论大小写如何)， 那么该元素的值就是 NULL 。而引号和反斜杠可以表示输入文本字符串"NULL"值。 另外，为了兼容 8.2 之前的版本，可以将array_nulls 配置参数设为 `off` 以禁止将 `NULL` 识别为 NULL 。

如前所示，当书写一个数组值的时候，可以在任何元素值周围使用双引号。 当元素值可能让数组值解析器产生歧义时，你必须这么做。 例如：元素值包含花括号、逗号(或者数据类型分割符)、双引号、反斜杠、 在开头/结尾处有空白符、匹配 NULL 的字符串。要在元素值中包含双引号或反斜杠， 可以加一个前导反斜杠。当然，你也可以避免引用和使用反斜杠逃逸来保护任何可能引起语法混淆的字符。

你可以在左花括弧前面或者右花括弧后面写空白。 你还可以在任意独立的项字符串前面或者后面写空白。所有这些情况下， 这些空白都会被忽略。不过，在双引号包围的元素里面的空白， 或者是元素里被两边非空白字符包围的空白，都不会被忽略。

> **Note:** 请记住你在 SQL 命令里写的任何东西都将首先解释成一个字符串文本， 然后才是一个数组。这样就造成你所需要的反斜杠数量翻了翻。比如， 要插入一个包含反斜杠和双引号的 `text` 数组，你需要这么写：
>
> ```
> INSERT ... VALUES (E'{"\\\\","\\""}');
> ```
>
> 字符串文本处理器去掉第一层反斜杠，然后剩下的东西到了数组数值分析器的时候将变成 `{"\\","\""}`。接着，该字符串传递给 `text` 数据类型的输入过程， 分别变成 `\` 和 `"`。如果我们使用的数据类型对反斜杠也有特殊待遇， 比如 `bytea`， 那么我们可能需要在命令里放多达八个反斜杠才能在存储态的数组元素中得到一个反斜杠。 也可以用美元符界定(参阅Section 4.1.2.4)来避免双份的反斜杠。
>
> **Tip:** `ARRAY` 构造器语法(参阅Section 4.2.12) 通常比数组文本语法好用些，尤其是在 SQL 命令里写数组值的时候。在 `ARRAY` 里， 独立的元素值的写法和数组里没有元素时的写法一样。

8.16. 复合类型

复合类型表示一行或者一条记录的结构； 它实际上只是一个字段名和它们的数据类型的列表。PostgreSQL 允许像简单数据类型那样使用复合类型。比如，一个表的某个字段可以声明为一个复合类型。

8.16.1. 声明复合类型

下面是两个定义复合类型的简单例子：

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

语法类似于 `CREATE TABLE`， 只是这里只可以声明字段名字和类型； 目前不能声明约束(比如 `NOT NULL`)。请注意 `AS` 关键字是很重要的； 没有它，系统会认为这是一个不同的 `CREATE TYPE` 命令， 因此你会看到奇怪的语法错误。

定义了类型，我们就可以用它创建表：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

或者函数：

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

在你创建表的时候，也会自动创建一个复合类型，名字与表名字相同， 表示该表的行类型。比如，如果我们说过:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

那么，和前面相同的 `inventory_item` 复合类型也会作为副产品创建， 并且可以和上面一样使用。不过，需要注意当前实现的一个重要限制： 因为现在还没有对复合类型实现约束，所以在表定义中显示的约束并不适用 于表之外的复合类型值。（一个部分绕开的办法是使用域类型作为复合类型的成员。）

8.16.2. 复合类型值输入

要以文本常量书写复合类型值，在圆括弧里包围字段值并且用逗号分隔他们。 你可以在任何字段值周围放上双引号，如果值本身包含逗号或者圆括弧， 你必须用双引号括起(更多细节见下面)。因此，复合类型常量的一般格式如下：

```
'( _val1_ , _val2_ , ... )'
```

一个例子是:

```
'("fuzzy dice",42,1.99)'
```

如果 `inventory_item` 类型在前面已经定义了，那么这是一个合法的数值。 要让一个字段值是 NULL，那么在列表里它的位置上不要写任何字符。比如， 下面这个常量在第三个字段声明一个 NULL ：

```
'("fuzzy dice",42,)'
```

如果你想要一个空字符串，而不是 NULL，写一对双引号：

```
'("",42,)'
```

这里的第一个字段是一个非 NULL 的空字符串，第三个字段是 NULL

(这些常量实际上只是我们在Section 4.1.2.7 讨论的一般类型常量的一个特殊例子。这些常量一开始只是当作字符串， 然后传递给复合类型输入转换器。一个明确的类型声明可能是必须的。)

我们也可以用 `ROW` 表达式语法来构造复合类型值。 在大多数场合下，这种方法都比用字符串文本的语法更简单，因为你不用操心多重引号。 我们已经在上面使用了这种方法了：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

只要你在表达式里有超过一个字段，那么关键字 ROW 就实际上是可选的， 所以可以简化为:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

`ROW` 表达式语法在Section 4.2.13里有更详细的讨论。

8.16.3. 访问复合类型

要访问复合类型字段的一个域，我们写出一个点以及域的名字， 非常类似从一个表名字里选出一个字段。实际上，因为实在太像从表名字中选取字段， 所以我们经常需要用圆括弧来避免分析器混淆。比如，你可能需要从 `on_hand` 例子表中选取一些子域，像下面这样：

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

这样将不能工作，因为根据 SQL 语法， `item` 是从一个表名字选取的， 而不是一个字段名字。你必须像下面这样写：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

或者如果你也需要使用表名字(比如，在一个多表查询里)，那么这么写：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

现在圆括弧对象正确地解析为一个指向 `item` 字段的引用，然后就可以从中选取子域。

类似的语法问题适用于在任何地点从一个复合类型值中查询一个域。比如， 要从一个返回复合类型值的函数中只选取一个字段，你需要写像下面这样的东西：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括弧，会产生一个语法错误。

8.16.4. 修改复合类型

下面是一些插入和更新复合类型字段的正确语法。首先，插入或者更新整个字段：

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

第一个例子省略了 `ROW`， 第二个使用它；两种方法都行。

我们可以更新一个复合字段的独立子域：

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

请注意，这里我们不需要(实际上是不能)在 `SET` 后面出现的字段名周围放上圆括弧， 但是我们在等号右边的表达式里引用同一个字段的时候却需要圆括弧。

我们也可以声明子域是 `INSERT` 的目标：

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

如果我们没有为字段的所有子域提供数值，那么剩下的子域将用 NULL 填充。

8.16.5. 复合类型输入和输出语法

一个复合类型的文本表现形式包含那些根据独立的子域类型各自 I/O 转换规则解析的项， 加上一些表明这是复合结构的修饰。这些修饰由整个数值周围的圆括弧(`(` 和 `)`)加上相邻域之间的逗号(`,`)组成。圆括弧外面的空白被忽略， 但是在圆括弧里面，它被当作子域数值的一部分，根据对该子域数据类型的输入转换规则， 这些空白可能有用，也可能没用。比如，在：

```
'(  42)'
```

里，如果子域类型是整数，那么空白将被忽略，但是如果是文本，那么就不会忽略。

如前面显示的那样，在给一个复合类型写数值的时候，你可以在独立的子域数值周围用双引号包围。 如果子域数值会导致复合数值分析器产生歧义，那么你必须这么做。 特别是子域包含圆括弧、逗号、双引号或反斜杠的场合。要想在双引号括起来的复合字段值里面放双引号或反斜杠， 那么你需要在它前面放一个反斜杠。（同样，在一个双引号括起的子域数值里面的一对双引号表示一个双引号字符， 就像 SQL 字符串文本的单引号规则一样。）另外， 你可以避免引用和用反斜杠逃逸的方法保护所有可能会当作复合类型语法的数据字符。

一个完全空的子域数值(在逗号或者逗号与圆括弧之间没有字符)表示一个 NULL 。 要写一个空字符串，而不是一个 NULL ，写 `""` 。

假如子域数值是空字符串或者包含圆括弧、逗号、双引号、反斜杠、空白， 复合类型输出程序会在子域数值周围放上双引号。（这么处理空白不是必须的， 但是可以增强易读性。）在一个子域数值里面嵌入的双引号和反斜杠将会写成两份。

> **Note:** 请注意你写的任何 SQL 命令都首先被当作字符串文本解析，然后才当作复合类型。 这就加倍了你需要的反斜杠数目(假设使用逃逸字符串的语法)。比如， 要插入一个包含双引号和一个反斜杠的 `text` 子域到一个复合类型数值里，你需要写：
>
> ```
> INSERT ... VALUES (E'("\\"\\\\")');
> ```
>
> 字符串文本处理器先去掉一层反斜杠，这样到达复合类型分析器的东西将变成 `("\"\\")`。 接着，该字符串传递给 `text` 数据类型的输入过程，变成 `"\`。 （如果我们使用的数据类型对反斜杠也有特殊待遇，比如 `bytea`， 那么我们可能需要在命令里放多达八个反斜杠以获取在存储的复合类型子域中有一个反斜杠。） 美元符界定(参阅Section 4.1.2.4)可以用于避免双份反斜杠的问题。
>
> **Tip:** 在 SQL 命令里写复合类型值的时候， `ROW` 构造器语法通常比复合文本语法更容易使用。 在 `ROW` 里，独立的子域数值的写法和并非作为复合类型的成员书写的方法一样。

8.17. 范围类型

范围数据类型代表着某一元素类型在一定范围内的值。 (此元素类型称为该范围的 子类型)。例如， `timestamp` 范围可能被用于代表一间会议室被预定的时间范围。 这种情况下数据 类型为 `tsrange` ("timestamp range"的简写)，并且 `timestamp` 是子类型。子类型必须具备完整的排序，这样清晰定义了元素值在范围之内，之前，或者之后。

范围类型是有用的。因为他们代表了在单一范围内的许多元素值，并且清晰表达了诸如重叠范围等概念。 出于计划目的的时间和日期范围的使用是一个最清晰的例子；价格范围， 仪器测量的范围等也有用。

8.17.1. 内嵌范围类型

PostgreSQL 提供下列内嵌范围类型：

- `int4range` — `integer` 的范围
- `int8range` — `bigint` 的范围
- `numrange` — `numeric` 的范围
- `tsrange` — `timestamp without time zone` 的范围
- `tstzrange` — `timestamp with time zone` 的范围
- `daterange` — `date` 的范围

此外，你可以定义你自己的范围类型；更多信息见CREATE TYPE。

8.17.2. 范例

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 包含
SELECT int4range(10, 20) @> 3;

-- 重叠
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 提取上边界
SELECT upper(int8range(15, 25));

-- 计算交叉
SELECT int4range(10, 20) * int4range(15, 25);

-- 范围是否为空
SELECT isempty(numrange(1, 5));
```

范围类型的操作数和函数的完整列表见Table 9-44 及Table 9-45。

8.17.3. 包含及不包含边界

每个非空范围有两个边界，下边界和上边界。这两个值之间的所有点都包含在范围内。 包含边界意味着边界点本身包含在范围内，而不包含边界意味着边界点不包含在范围内。

在一个文本格式的范围里，包含下边界由"`[`"代表，而不包含下边 界由"`(`"代表。同样，一个包含上边界由"`]`"代表，而不包含上边界由"`)`"代表。（更多细节见Section 8.17.5）

函数 `lower_inc` 和 `upper_inc` 分别检测一个范围值的上下边界 是否包含。

8.17.4. 无限（无边界）范围

一个范围的下边界可以被省略，意味着小于上边界的所有点都包含在范围内。同样，如果范围的上边界 被省略，那么所有大于下边界的点都包含在范围内。如果上下边界都被省略，那么所有元素类型的值都 被认为在范围内。

相当于分别认为下边界是"负无穷大"，或者上边界是"正无穷大"。但是注意 这些无穷大值绝不是范围元素类型的值，而且绝不是范围的一部分。（所以没有包含无穷边界之类的东西 — 如果你尝试写一个，它会被自动转换成一个不包含边界。）

某些元素类型还有一个"无限"的概念，但是只要涉及到范围类型机制它就只是另一个值。例如， 在timestamp范围里， `[today,]` 意味着和 `[today,)` 是相同的东西。但是 `[today,infinity]` 意味着与 `[today,infinity)` 不同的东西 — 后者不 包含特殊的 `timestamp` 值 `infinity` 。

函数 `lower_inf` 和 `upper_inf` 分别检测一个范围的无限下边界 和上边界。

8.17.5. 范围输入/输出

范围值的输入必须遵循下面的格式：

```
(_下边界_,_上边界_)
(_下边界_,_上边界_]
[_下边界_,_上边界_)
[_下边界_,_上边界_]
空
```

如前所述，圆括号或者方括号显示下边界和上边界是不包含的还是包含的。注意最后的格式是 `空`，代表着一个空的范围（一个不含有值的范围）。

`_下边界_` 可以是子类型有效输入的一个字符串，或者是空以显示没有下边 界。同样，`_上边界_` 可以是子类型有效输入的一个字符串，或者是空以显 示没有上边界。

每个边界值可以用 `"` （双引号）字符引用。如果边界值包含圆括号，方括号，逗号， 双引号，或者反斜杠，这就很有必要。因为不这样的话，这些字符会被当成范围语法的一部分。要想把 双引号或反斜杠放入一个引用的边界值，就在它前面加一个反斜杠。（另外，加了双引号的边界值内 的两个连续的双引号用来表示一个双引号字符，类似于SQL文字串内的单引号规则。）或者，你可以避免用引号，使 用反斜杠转义来保护所有数据字符不被认为是范围语法。而且，要写入一个空字符串边界值，用 `""` 。这是因为什么也不写入意味着无限边界。

在范围值前后可以有空格，但是在圆括号和方括号之间的任何空格都被认为是上边界或下边界的一部分。 （重要还是不重要取决于元素类型。）

> **Note:** 这些规则很类似于在复合类型常量中写入字段值。更多注释见Section 8.16.5。

例子：

```
-- 包括3，不包括7，并且包括二者之间的所有点
SELECT '[3,7)'::int4range;

-- 不包括3和7，但是包括二者之间所有点
SELECT '(3,7)'::int4range;

-- 只包括单一值4
SELECT '[4,4]'::int4range;

-- 不包括点（被标准化为‘空’）
SELECT '[4,4)'::int4range;
```

8.17.6. 构造范围

每个范围有一个与范围类型同名的构造函数。使用构造函数往往比写入一个范围文本常量更便利，因为 它避免了额外引用边界值的需要。构造函数接受两到三个参数。两参数方式构造一个标准格式的范围（ 包含下边界，不包含上边界），而三参数方式用第三个参数指定边界

来构造范围。 第三个参数必须是下面的字符串之一 "`()`", "`(]`", "`[)`", 或 "`[]`". 例如：

```
-- 完整方式为：下边界, 上边界, 和指示包含还是不包含边界的文本参数
SELECT numrange(1.0, 14.0, '(]');

-- 如果第三个参数省略, 使用'[)'。
SELECT numrange(1.0, 14.0);

-- 尽管在这里指定了'(]', 然而该值会被转换成标准格式。这是由于int8range是一个离散范围类型
（见下面）。
SELECT int8range(1, 14, '(]');

-- 使用NULL作任一边界会导致范围在那一边没有边界。
SELECT numrange(NULL, 2.2);
```

8.17.7. 离散范围类型

离散范围的元素类型有一个完善定义的"阶梯", 例如 `integer` 或者 `date` 。当在这些类型里两个元素中间没有有效值时, 它们可被称为是邻近的。与之形成 对比的是连续范围, 总是（或者几乎总是）可能在两个给定值之间找到其它元素。例如, `numeric` 类型的范围是连续的, 和 `timestamp` 一样。（即使 `timestamp` 精度 有限, 理论上可以被当做是离散的, 但是既然不关心阶梯大小最好还是把它当做是连续的。）

考虑一个离散范围的另一种方式是每一个元素值都清晰知道它的"下一个"或者"上一个" 值。知道了这个, 通过选择下一个或者上一个元素值而不是开始给定的值, 就可能在一个范围边界的包 含和不包含表达之间进行转换。例如, 在一个整数范围类型里 `[4,8]` 和 `(3,9)` 提供相同的值集合；但是数值范围不是这样。

离散范围应当有一个标准化函数, 该函数知道元素类型想要的阶梯大小。这个标准化函数 负责把该范围类型的等效值转换成同一表达方式, 尤其是包含或不包含边界。如果不指定一个标准化函数, 不同格式的范围会被认为是不相等的, 即使它们实际上可能代表相同的值集合。

内嵌范围类型 `int4range` , `int8range` , 和 `daterange` 都使用 包括下边界不包含上边界的标准格式；即, `[)` 。然而用户定义的范围类型可以使用 其他规则。

8.17.8. 定义新的范围类型

用户可以定义他们自己的范围类型。这么做通常是为了使用内嵌范围类型所不提供的子类型范围。 例如, 定义一个新的范围子类型 `float8`：

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

因为 `float8` 没有有意义的"阶梯"，我们在此例中不定义一个标准化函数。

如果子类型被认为含有离散的而不是连续的值，命令 `CREATE TYPE` 应当指定一个 `canonical` (标准化)函数。 标准化函数使用一个输入范围值，并且必须返回一个可能含有不同边界和格式的相应的范围值。代表相同 值集合的两个范围的输出必须相同，比如整数范围 `[1, 7]` 和 `[1,8)` 。 你选择哪个表达方式作为标准没有关系，只要两个内容相当但格式不同的值总是映射到相同格 式的相同值。除了调整包含/不包含边界格式以外， 一旦想要的阶梯大小大于子类型能够储存的范围， 标准化函数可能取整边界值。例如， `timestamp` 范围类型可以被定义为以一个小时作为阶梯值。这 种情况下标准化函数需要把不是一小时的整数倍的值化成一小时的整数倍，或者可能抛出一个错误。

定义你自己的范围类型也允许你指定使用一个不同的子类型B-tree操作符类或排序规则，以便改变排序次序 来决定哪些值落入一个给定的范围。

此外，任何打算要使用GiST或SP-GiST索引的范围类型应当定义一个子类型差异，或者 `subtype_diff` 函数。（没有 `subtype_diff` 索引仍然可以起作用，但是比起提 供差异函数时可能相当低效。）子类型差异函数采用子类型的两个输入值，并返回它们之间表示为 `float8` 值的差异（就是说， _X_ 减去 _Y_ ）。在我们上面的例子 中，可以使用常规 `float8` 减操作符调用的函数；但是对其它子类型， 类型转换似乎是必要的。 关于怎样将差异表示为数字的某些创新想法可能也是必要的。 `subtype_diff` 函数应当尽最 大可能与所选操作符类和排序规则表明的排序次序相一致；即，每当根据排序次序第一个参数大于第 二个参数时，结果应当是正数。

更多创建范围类型的信息见CREATE TYPE。

8.17.9. 索引

可以对范围类型的表列创建GiST和SP-GiST索引。例如，要创建一个GiST索引：

```
CREATE INDEX reservation_idx ON reservation USING gist (during);
```

一个GiST或者SP-GiST索引可以加速包含这些范围操作符的查询： `=` , `&&` , `<@` , `@>` , `<<` , `>>` , `-|-` , `&<` , and `&>` （更多信息见Table 9-44）。

此外，对范围类型的表列可以创建B-tree和哈希索引。对这些索引类型，基本上唯一可用的范围操作是等 于。用相应的 `<` 和 `>` 操作符可以为范围索引定义一个B-tree排序次序，但是这个次序相当武断，在现实世界中通常没有用。范围类型的B-tree和哈希支持主要是用于查询内部的排 序和哈希操作，而不是用于实际索引的创建。

8.17.10. 范围上的约束

当 `UNIQUE` 是一个对标量值的自然约束时，对范围类型通常是不合适的。反而不包含 约束往往更合适（见CREATE TABLE ... CONSTRAINT ... EXCLUDE）。不包含约束允许对一个范围类型指定约束，比如 "非重叠"。例如：

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING gist (during WITH &&)
);
```

这个约束将会防止任何重叠的值同时存在于表中：

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

你可以使用 `btree_gist` 扩展对简单标量数据类型 定义不包含约束。简单标量数据类型可以和范围不包含结合来获得最大的灵活性。例如， 在 `btree_gist` 安装后，下列的约束排除重叠的范围，除非会议室房间号相等：

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING gist (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflic
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. 对象标识符类型

PostgreSQL在内部使用对象标识符(OID)作为各种系统表的主键。 同时，系统不会给用户创建的表增加一个 OID 系统字段(除非在建表时声明了 `WITH OIDS` 或者配置参数default_with_oids设置为开启)。 `oid` 类型代表一个对象标识符。除此以外 `oid` 还有几个别名： `regproc` , `regprocedure` , `regoper` , `regoperator` , `regclass` , `regtype` , `regconfig` , 和 `regdictionary` 。 Table 8-23显示了概览。

目前 `oid` 类型用一个四字节的无符号整数实现。因此， 它不够提供大数据库范围内的唯一性保证，甚至在单个的大表中也不行。 因此，我们不鼓励在用户创建的表中使用 OID 字段做主键。OID 最好只是用于系统表。

`oid` 类型本身除了比较之外还有几个操作。不过，它可以转换为整数， 然后用标准的整数操作符操作。如果你这么干，请注意可能的有符号和无符号之间的混淆。

OID 别名类型除了输入和输出过程之外没有自己的操作。 这些过程可以为系统对象接受和显示符号名，而不仅仅是类型 `oid` 将要使用的行数值。别名类型允许我们简化为对象查找 OID 值的过程。比如， 检查和一个表 `mytable` 相关的 `pg_attribute` 行，我们可以这样写:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

而不用:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

虽然看上去不坏，但是这个例子还是简化了好多，如果在不同的模式里有好多叫 `mytable` 的表，那么我们需要写一个更复杂的子查询。 `regclass` 的输入转换器处理根据模式路径设置的表检索工作，所以它自动干了"正确的事情"。 类似的还有，把一个表的 OID 转换成 `regclass` 是查找一个 OID 对应的符号名称的最简单方法。

Table 8-23. 对象标识符类型

名字	引用	描述	数值例子
oid	任意	数字化的对象标识符	564182
regproc	pg_proc	函数名字	sum
regprocedure	pg_proc	带参数类型的函数	sum(int4)
regoper	pg_operator	操作符名	+
regoperator	pg_operator	带参数类型的操作符	*(integer,integer) 或 -(NONE,integer)
regclass	pg_class	关系名	pg_type
regtype	pg_type	数据类型名	integer
regconfig	pg_ts_config	文本搜索配置	english
regdictionary	pg_ts_dict	文本搜索字典	simple

所有 OID 别名类型都接受有模式修饰的名字， 并且如果在当前搜索路径中不增加修饰无法找到该对象的话， 那么在输出时将显示有模式修饰的名字。 regproc 和 regoper 别名类型将只接受唯一的输入名字(不能重载)，因此它们的用途有限。 对于大多数应用， regprocedure 或 regoperator 更合适。 对于 regoperator ， 单目操作符是通过在那些未用的操作数上写 NONE 来标识的。

OID 别名类型的一个额外的属性是依赖关系的创建。 如果这些类型之一的常量出现在一个存储的表达式里(比如字段缺省表达式或者视图)， 它在被引用的对象上创建一个依赖性。比如，如果一个字段有缺省的 nextval('my_seq'::regclass) 表达式，PostgreSQL 理解缺省表达式依赖于序列 my_seq ；系统将不允许在删除缺省的表达式之前删除该序列。

系统使用的另外一个标识符类型是事务(缩写<abbr>xact</abbr>)标识符 xid 。 它是系统字段 xmin 和 xmax 的数据类型。事务标识符是 32 位的量。

系统需要的第三种标识符类型是命令标识符 cid 。 它是系统字段 cmin 和 cmax 的数据类型。命令标识符也是 32 位的量。

系统使用的最后一个标识符类型是行标识符 tid 。 它是系统表字段 ctid 的数据类型。行 ID 是一对数值(块号，块内的行索引)， 它标识该行在其所在表内的物理位置。

系统字段在Section 5.4里有更多解释。

8.19. 伪类型

PostgreSQL类型系统包含一系列特殊用途的条目， 它们按照类别来说叫做伪类型。伪类型不能作为字段的数据类型， 但是它可以用于声明一个函数的参数或者结果类型。 伪类型在一个函数不只是简单地接受并返回某种SQL 数据类型的情况下很有用。Table 8-24列出了所有的伪类型。

Table 8-24. 伪类型

名字	描述
`any`	表示一个函数接受任何输入数据类型。
`anyelement`	表示一个函数接受任何数据类型 (参阅Section 35.2.5)。
`anyarray`	表示一个函数接受任意数组数据类型 (参阅Section 35.2.5)。
`anynonarray`	表示一个函数接受任意非数组数据类型 (参阅Section 35.2.5)。
`anyenum`	表示一个函数接受任意枚举数据类型 (参阅Section 35.2.5 和 Section 8.7)。
`anyrange`	表示一个函数接受任意范围数据类型 (参阅 Section 35.2.5 和 Section 8.17)。
`cstring`	表示一个函数接受或者返回一个空结尾的 C 字符串。
`internal`	表示一个函数接受或者返回一种服务器内部的数据类型。
`language_handler`	一个过程语言调用处理器声明为返回 `language_handler` 。
`fdw_handler`	一个外部数据封装器声明为返回 `fdw_handler` 。
`record`	标识一个函数返回一个未声明的行类型。
`trigger`	一个触发器函数声明为返回 `trigger` 。
`void`	表示一个函数不返回数值。
`opaque`	一个已经过时的类型，以前用于所有上面这些用途。

用 C 编写的函数(不管是内置的还是动态装载的)都可以声明为接受或者返回这样的伪数据类型。 在把伪类型用做函数参数类型的时候，保证函数行为正常就是函数作者的任务了。

用过程语言编写的函数只能根据它们的实现语言是否可以使用伪类型而使用它。 目前，过程语言都不允许使用伪类型作为参数类型，并且只允许使用 `void` 和 `record` 作为结果类型(如果函数用做触发器，那么加上 `trigger`)。 一些多态的函数还支持使用 `anyelement` ， `anyarray` ， `anynonarray` `anyenum` 和 `anyrange` 类型。

伪类型`internal`用于声明那种只能在数据库系统内部调用的函数， 它们不能直接在SQL查询里调用。如果函数至少有一个 `internal` 类型的参数，那么我们就不能从SQL 里调用它。为了保留这个限制的类型安全，我们一定要遵循这样的编码规则： 不要创建任何声明为返回 `internal` 的函数， 除非它至少有一个`internal`参数。

Chapter 9. 函数和操作符

Table of Contents

PostgreSQL为内建的数据类型提供了大量的函数和操作符。 用户也可以定义它们自己的函数和操作符，像Part V 里描述的那样。psql命令 `\df` 和 `\do` 可以分别用于列出所有实际可用的函数和操作符。

如果你关心移植性，那么请注意，我们在本章描述的大多数函数和操作符， 除了最琐碎的算术和比较操作符以及一些做了明确标记的函数以外， 都没有在SQL标准里声明。许多其它SQL 实现也有这些扩展的功能，并且很多时候不同的数据库产品中这些功能是相互兼容的。本章也并没有穷尽一切信息；一些附加的函数在本手册的相关章节里出现。

9.1. 逻辑操作符

常用的逻辑操作符有：

`AND`
`OR`
`NOT`

SQL使用三值的逻辑体系，真，假和 `null`， 这时 `null` 代表"未知"。观察下面真值表：

`_a_`	`_b_`	`_a_` AND `_b_`	`_a_` OR `_b_`
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

`_a_`	NOT `_a_`
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符 `AND` 和 `OR` 都是可交换的，也就是说， 你可以交换左右操作数而不影响结果。但是请参阅Section 4.2.14 获取有关子表达式计算顺序的更多信息。

9.2. 比较操作符

可用的比较操作符在Table 9-1显示。

Table 9-1. 比较操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于
<> 或 !=	不等于

> **Note:** != 操作符在分析器阶段被转换成 <> 。 != 和 <> 操作符是完全等价的。

比较操作符可以用于所有相关的数据类型。所有比较操作符都是双目操作符， 返回 boolean 类型数值；像 1 < 2 < 3 这样的表达式是非法的(因为布尔值和 3 之间不能做比较)。

除了比较操作符以外，我们还可以使用 BETWEEN 构造。

```
_a_ BETWEEN _x_ AND _y_
```

等效于

```
_a_ >= _x_ AND _a_ <= _y_
```

注意 BETWEEN 认为端点值是包含在范围内的。 NOT BETWEEN 做相反的比较：

```
_a_ NOT BETWEEN _x_ AND _y_
```

等效于

```
_a_ < _x_ OR _a_ > _y_
```

`BETWEEN SYMMETRIC` 和 `BETWEEN` 一样，只是没有要求 `AND` 左边的参数小于或等于右边的参数。如果左面的参数不小于或等于右面的参数， 那么两个参数是自动交换的，所以非空范围总是适用。

要检查一个值是否为 NULL，使用下面的构造：

```
_expression_ IS NULL
_expression_ IS NOT NULL
```

或者等效，但并不标准的构造：

```
_expression_ ISNULL
_expression_ NOTNULL
```

不要写 `_expression_` = NULL 因为 `NULL` 是不"等于" `NULL` 的。 NULL 代表一个未知的数值，因此我们无法知道两个未知的数值是否相等。这个行为遵循 SQL 标准。

> **Tip:** 有些应用可能要求表达式 `_expression_` = NULL 在 `_expression_` 为 NULL 时候返回真。 我们强烈建议这样的应用修改成遵循 SQL 标准。但是，如果这样修改是不可能的，那么我们可以打开transform_null_equals配置参数， 让PostgreSQL将 `x = NULL` 自动转换成 `x IS NULL` 。
>
> **Note:** 如果 `_expression_` 是行值， 那么当行表达式本身为 NULL 或该行的所有字段都为 NULL 时， `IS NULL` 将为真； 当行表达式本身不为 NULL 并且该行的所有字段都不为 NULL 时， `IS NOT NULL` 也将为真。因为这个行为， `IS NULL` 和 `IS NOT NULL` 并不总是为行值表达式返回相反的值， 也就是， 一个同时包含NULL和non-null值的行值表达式将在两种情况下都返回false。 这个规定符合 SQL 标准，但是与PostgreSQL之前的版本不兼容。

如果有任何一个输入是 NULL， 那么普通的比较操作符生成 NULL(表示"未知")， 而不是true或false。例如， `7 = NULL` 生成null， `7 <> NULL` 也生成null。当这种行为不适用时，使用 `IS [ NOT ] DISTINCT FROM` 构造：

```
_expression_ IS DISTINCT FROM _expression_
_expression_ IS NOT DISTINCT FROM _expression_
```

对于非 NULL 的输入 `IS DISTINCT FROM` 与 `<>` 操作符相同。但是，如果两个输入都是 NULL， 那么它将返回假；如果只有一个输入是 NULL， 那么它将返回真。类似的， 对于非 NULL 的输入 `IS NOT DISTINCT FROM` 与 `=` 操作符相同。但是，如果两个输入都是 NULL， 那么它将返回真； 如果只有一个输入是 NULL， 那么它将返回假。这样就很有效地把 NULL 当作一个普通数据值看待， 而不是"未知"。

布尔数值可以用下面的构造进行测试

```
_expression_ IS TRUE
_expression_ IS NOT TRUE
_expression_ IS FALSE
_expression_ IS NOT FALSE
_expression_ IS UNKNOWN
_expression_ IS NOT UNKNOWN
```

这些构造将总是返回真或假，从来不返回 NULL ，即使操作数是 NULL 也如此。 NULL 输入被当做逻辑数值"未知"。请注意实际上 `IS UNKNOWN` 和 `IS NOT UNKNOWN` 分别与 `IS NULL` 和 `IS NOT NULL` 相同，只是输入表达式必须是布尔类型。

9.3. 数学函数和操作符

PostgreSQL为许多类型提供了数学操作符。 对于那些没有标准的数学传统的类型(比如日期/时间类型)， 我们在随后的章节里描述实际的行为。

Table 9-2显示了可用的数学操作符。

Table 9-2. 数学操作符

操作符	描述	例子	结果
+	加	2 + 3	5
-	减	2 - 3	-1
*	乘	2 * 3	6
/	除(整数除法将截断结果)	4 / 2	2
%	模(求余)	5 % 4	1
^	幂(指数运算)	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘	5 !	120
!!	阶乘(前缀操作符)	!! 5	120
@	绝对值	@ -5.0	5
&	二进制 AND	91 & 15	11
		二进制 OR	32	3	35
#	二进制 XOR	17 # 5	20
~	二进制 NOT	~1	-2
<<	二进制左移	1 << 4	16
>>	二进制右移	8 >> 2	2

位操作符只能用于整数类型，而其它的操作符可以用于全部数值类型。 位操作符还可以用于位串类型 `bit` 和 `bit varying` ， 如Table 9-10所示。

Table 9-3显示了可用的数学函数。在该表中， `dp` 表示 `double precision` 。 这些函数中有许多都有多种不同的形式，区别是参数不同。除非特别指明， 任何特定形式的函数都返回和它的参数相同的数据类型。处理 `double precision` 数据的函数大多数是在宿主系统的C库的基础上实现的；因此， 精度和数值范围方面的行为是根据宿主系统而变化的。

Table 9-3. 数学函数

函数	返回类型	描述	例子
`abs(```_x_)	(与输入相同)	绝对值	`abs(-17.4)`
`cbrt(```dp)	dp	立方根	`cbrt(27.0)`
`ceil(```dp 或 numeric)	(与输入相同)	不小于参数的最小的整数	`ceil(-42.8)`
`ceiling(```dp 或 numeric)	(与输入相同)	不小于参数的最小整数 (`ceil` 的别名)	`ceiling(-95.3)`
`degrees(```dp)	`dp`	把弧度转为角度	`degrees(0.5)`
`div(```y numeric , x numeric)	numeric	integer quotient of y / x	`div(9,4)`
`exp(```dp 或 numeric)	(与输入相同)	自然指数	`exp(1.0)`
`floor(```dp 或 numeric)	(与输入相同)	不大于参数的最大整数	`floor(-42.8)`
`ln(```dp 或 numeric)	(与输入相同)	自然对数	`ln(2.0)`
`log(```dp 或 numeric)	(与输入相同)	以 10 为底的对数	`log(100.0)`
`log(```b numeric , x numeric)	`numeric`	以 b 为底数的对数	`log(2.0, 64.0)`
`mod(```y , x)	(与参数类型相同)	y / x 的余数(模)	`mod(9,4)`
`pi()`	`dp`	"π" 常量	`pi()`
`power(```a dp , b dp)	dp	a 的 b 次幂	`power(9.0, 3.0)`
`power(```a numeric , b numeric)	`numeric`	a 的 b 次幂	`power(9.0, 3.0)`
`radians(```dp)	dp	把角度转为弧度	`radians(45.0)`
`random()`	`dp`	0.0 到 1.0 之间的随机数	`random()`
`round(```dp 或	(与输入	圆整为最接近	

numeric)	相同)	的整数	round(42.4)
`round(```v numeric , s int)	numeric	圆整为 s 位小数	round(42.4382, 2)
`setseed(```dp)	void	为随后的 random() 调用设置种子(-1.0 到 1.0 之间，包含)	setseed(0.54823)
`sign(```dp 或 numeric)	(与输入相同)	参数的符号(-1, 0, +1)	sign(-8.4)
`sqrt(```dp 或 numeric)	(与输入相同)	平方根	sqrt(2.0)
`trunc(```dp 或 numeric)	(与输入相同)	截断(向零靠近)	trunc(42.8)
`trunc(```v numeric , s int)	numeric	截断为 s 位小数	trunc(42.4382, 2)
`width_bucket(```op numeric , b1 numeric , b2 numeric , count int)	int	返回一个桶，这个桶是在一个有 count 个桶，上界为 b1 下界为 b2 的等深柱图中 operand 将被赋予的那个桶。	width_bucket(5.35, 0.024, 10.06,
`width_bucket(```op dp , b1 dp , b2 dp , count int)	int	返回一个桶，这个桶是在一个有 count 个桶，上界为 b1 下界为 b2 的等深柱图中 operand 将被赋予的那个桶。	width_bucket(5.35, 0.024, 10.06,

最后，Table 9-4显示了可用的三角函数。 所有三角函数都使用类型为 double precision 的参数和返回类型。 三角函数参数用弧度来表达。反函数的返回值也是用弧度来表达的。 参阅上面的单元转换函数 radians() 和 degrees() 。

Table 9-4. 三角函数

函数	描述
`acos(```_x_)	反余弦
`asin(```_x_)	反正弦
`atan(```_x_)	反正切
`atan2(```_y_ , _x_)	_y_ / _x_ 的反正切
`cos(```_x_)	余弦
`cot(```_x_)	余切
`sin(```_x_)	正弦
`tan(```_x_)	正切

9.4. 字符串函数和操作符

本节描述了用于检查和操作字符串数值的函数和操作符。 在这个环境中的字符串包括`character`，`character varying`，`text`类型的值。除非另外说明，所有下面列出的函数都可以处理这些类型， 不过要小心的是，在使用`character`类型的时候，需要注意自动填充的潜在影响。 有些函数还可以处理位串类型。

SQL定义了一些字符串函数，用特定的关键字而不是逗号来分隔参数。 详情请见Table 9-5。PostgreSQL 也提供了使用正常的函数调用语法实现的这些函数的版本(参阅 Table 9-6)。

> **Note:** 在PostgreSQL 8.3之前， 这些函数将默默接受一些非字符串数据类型的值，由于存在从这些数据类型到`text`的隐式强制转换，转换后的它们经常发生意外的行为，因此删除了隐式强制转换。 然而，字符串连接操作符(`||`)仍接受非字符串输入， 只要至少有一个输入是字符串类型，如Table 9-5所示。 对于其它情况下，如果你需要重复以前的行为，插入一个明确的强制转换到`text`。

Table 9-5. SQL 字符串函数和操作符

函数	返回类型	描述
`string` `||` `string`	`text`	字符串连接
`string` `||` `non-string` 或 `non-string` `||` `string`	`text`	带有一个非字符串输入的字符串连接
`bit_length(```string`)	`int`	字符串的位
`char_length(```string`)` 或 character_length(``string`)	`int`	字符串中的字符个数
`lower(```string`)	`text`	把字符串转化为小写
`octet_length(```string`)	`int`	字符串中的字节数
`overlay(```string` placing `string` from `int` [for `int`])	`text`	替换子字符串
`position(```substring` in `string`)	`int`	指定子字符串的位置
`substring(```string` [from `int` ] [for `int`])	`text`	截取子字符串
`substring(```string` from `_pattern_`)	`text`	截取匹配POSIX正则表达式的子字符串。参阅Section 9.7获取更多关于模式匹配的信息。
`substring(```string` from `_pattern_` for `_escape_`)	`text`	截取匹配SQL正则表达式的子字符串。 参阅Section 9.7获取更多关于模式匹配的信息。
`trim([leading | trailing | both] [```characters` ] from `string`)	`text`	从字符串 `string` 的开头/结尾/两边 删除只包含 `characters` 中字符 (缺省是空白)的最长的字符串
`upper(```string`)	`text`	把字符串转化为大写

还有额外的字符串操作函数可以用，它们在Table 9-6列出。 它们有些在内部用于实现Table 9-5 列出的SQL标准字符串函数。

Table 9-6. 其它字符串函数

函数	返回类型	描述	
`ascii(``string)	int	参数中第一个字符的ASCII编码值。 对于UTF8返回字符的宽字节编码值。 对于其它的多字节编码，参数必须是一个ASCII字符。	ascii
`btrim(``string text [, characters text])	text	从 string 开头和结尾删除只包含 characters 中字符(缺省是空白)的最长字符串。	btrim
`chr(``int)	text	给定编码的字符。 对于UTF8这个参数作为宽字节代码处理。 对于其它的多字节编码，这个参数必须指定一个ASCII字符， 因为text数据类型无法存储NULL数据字节，不能将NULL(0)作为字符参数。	chr(6
`concat(``str "any" [, str "any" [, ...]])	text	连接所有参数的文本表示。NULL 参数被忽略。	concat
`concat_ws(``sep text , str "any" [, str "any" [, ...]])	text	连接所有参数，但是第一个参数是分隔符，用于将所有参数分隔。NULL 参数被忽略。	concat
`convert(``string bytea , src_encoding name , dest_encoding name)	bytea	把原来编码为 src_encoding 的字符串转换为 dest_encoding 编码。 在这种编码格式中 string 必须是有效的。 用 CREATE CONVERSION 定义转换。 这也有些预定义的转换。 参阅Table 9-7显示可用的转换。	conver
`convert_from(``string bytea ,	text	把原来编码为 src_encoding 的字符串转换为数据库编码格	conver

src_encoding name)		式。 这种编码格式中， string 必须是有效的。	
`convert_to(```string text , dest_encoding name)	bytea	将字符串转化为 dest_encoding 编码格式。	conver
`decode(```string text , format text)	bytea	把用 string 表示的文本里面的二进制数据解码。 format 选项和 encode 相同。	decode
`encode(```data bytea , format text)	text	把二进制数据编码为文本表示。支持的格式有： base64 , hex , escape 。 escape 转换零字节和高位设置字节为八进制序列 (\``_nnn_) 和双反斜杠。	encode
format`( formatstr `text [, formatarg "any" [, ...]])	text	根据格式字符串格式参数。这个函数类似C函数 sprintf 。 参阅Section 9.4.1。	format
`initcap(```string)	text	把每个单词的第一个字母转为大写，其它的保留小写。 单词是一系列字母数字组成的字符，用非字母数字分隔。	initca
`left(```str text , n int)	text	返回字符串的前 _n_ 个字符。当 _n_ 是负数时， 返回除最后| _n_ |个字符以外的所有字符。	left(
`length(```string)	int	string 中字符的数目	length
`length(```string bytea , encoding name)	int	指定 encoding 编码格式的 string 的字符数。 在这个编码格式中， string 必须是有效的。	length
`lpad(```string text , length int [, fill text])	text	通过填充字符 fill (缺省时为空白)，把 string 填充为 length 长度。 如果 string 已经比 length 长则将其尾部截断。	lpad(
		从字符串 string 的开	

`ltrim(```string text [, characters text])	text	头删除只包含 characters 中字符(缺省是一个空白)的最长的字符串。	ltrim(
`md5(```string)	text	计算 string 的MD5散列，以十六进制返回结果。	md5('a
pg_client_encoding()	name	当前客户端编码名称	pg_cl
`quote_ident(```string text)	text	返回适用于SQL语句的标识符形式(使用适当的引号进行界定)。 只有在必要的时候才会添加引号(字符串包含非标识符字符或者会转换大小写的字符)。 嵌入的引号被恰当地写了双份。 又见Example 40-1。	quote_
`quote_literal(```string text)	text	返回适用于在SQL语句里当作文本使用的形式(使用适当的引号进行界定)。 嵌入的引号和反斜杠被恰当地写了双份。请注意，当输入是null时， quote_literal 返回null；如果参数可能为null， 通常 quote_nullable 更适用。 又见Example 40-1。	quote_
`quote_literal(```value anyelement)	text	将给定的值强制转换为text，加上引号作为文本。嵌入的引号和反斜杠被恰当地写了双份。	quote_
`quote_nullable(```string text)	text	返回适用于在SQL语句里当作字符串使用的形式(使用适当的引号进行界定)。 或者，如果参数为空，返回 NULL 。 嵌入的引号和反斜杠被恰当地写了双份。 又见Example 40-1。	quote_
`quote_nullable(```value anyelement)	text	将给定的参数值转化为text，加上引号作为文本；或者，如果参数为空， 返回 NULL 。嵌入的引号和反斜杠被恰当地写了双份。	quote_

`regexp_matches(```string text, pattern text [, flags text])	setof text[]	返回 string 中所有匹配POSIX正则表达式的子字符串。 参阅Section 9.7.3获得更多信息。	regexp
`regexp_replace(```string text , pattern text , replacement text [, flags text])	text	替换匹配 POSIX 正则表达式的子字符串。参见Section 9.7.3 以获取更多模式匹配的信息。	regexp
`regexp_split_to_array(```string text, pattern text [, flags text])	text[]	用POSIX正则表达式作为分隔符，分隔 string 。 参阅Section 9.7.3以获取更多模式匹配的信息。	regexp
`regexp_split_to_table(```string text , pattern text [, flags text])	setof text	用POSIX正则表达式作为分隔符，分隔 string 。 参阅Section 9.7.3以获取更多模式匹配的信息。	regexp

world

(2 rows) | |
repeat(```string` `text`, `number` `int`) | `text` | 将`string`重复`number`次 | `repeat('Pg'
replace(string` `text`, `from` `text`, `to` `text`) | `text` | 把字符串`string`里出现地所有子字
str) | text | 返回颠倒的字符串 | reverse('abcde') | edcba | | ``right(```str text , n int) | text | 返回字符串中的后 _n_ 个字符。当 _n_ 是负值时， 返回除前| _n_ |个字符以外的所有字符。 | right('abcde', 2) | de | |
rpad(```string` `text`, `length` `int` [, `fill` `text`]) | `text` | 使用填充字符`fill`(缺省时
rtrim(string` `text` [, `characters` `text`]) | `text` | 从字符串`string`的结尾删除只包含 `char
string `text , delimiter text , field int) | text | 根据 delimiter 分隔 string 返回生成的第 field 个子字符串(1为基)。 | split_part('abc~@~def~@~ghi', '~@~', 2) | def | |
strpos(```string`, `substring`) | `int` | 指定的子字符串的位置。和`position(substring in string)一样，不过参数顺序相反。 | strpos('high', 'ig') | 2 | | ``substr(```string , from [, count]) | text | 抽取子字符串。和 substring(``string from from for count))一样 |
substr('alphabet', 3, 2) | ph | |
to_ascii(```string` `text` [, `encoding` `text`]) | `text` | 把`string`从其它编码转换为ASCII
to_hex(number` `int` or `bigint`) | `text` | 把`number`转换成十六进制表现形式 | `to_hex(214748
string `text , from text , to text) | text | 把在 string 中包含的任何匹配 from 中字符的字符转化为对应的在 to 中的字符。 如果 from 比 to 长， 删掉在 from 中出现的额外的字符。 | translate('12345', '143', 'ax') | a2x5 |

concat , concat_ws 和 format 函数是可变的，所以用 VARIADIC 关键字标记传递的数值以连接或者格式化为一个数组是可能的。 见Section 35.4.5。数组的元素对函数来说是单独的普通参数。 如果可变数组的元素是NULL，那么 concat 和 concat_ws 返回NULL，但是 format 把

NULL作为零元素数组对待。

又见Section 9.20里面的聚集函数 `string_agg` 。

Table 9-7. 内置的转换

转换名 [a]	源编码	目的编码
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R

iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII

utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874

`windows_1250_to_iso_8859_2`	`WIN1250`	`LATIN2`
`windows_1250_to_mic`	`WIN1250`	`MULE_INTERNAL`
`windows_1250_to_utf8`	`WIN1250`	`UTF8`
`windows_1251_to_iso_8859_5`	`WIN1251`	`ISO_8859_5`
`windows_1251_to_koi8_r`	`WIN1251`	`KOI8R`
`windows_1251_to_mic`	`WIN1251`	`MULE_INTERNAL`
`windows_1251_to_utf8`	`WIN1251`	`UTF8`
`windows_1251_to_windows_866`	`WIN1251`	`WIN866`
`windows_1252_to_utf8`	`WIN1252`	`UTF8`
`windows_1256_to_utf8`	`WIN1256`	`UTF8`
`windows_866_to_iso_8859_5`	`WIN866`	`ISO_8859_5`
`windows_866_to_koi8_r`	`WIN866`	`KOI8R`
`windows_866_to_mic`	`WIN866`	`MULE_INTERNAL`
`windows_866_to_utf8`	`WIN866`	`UTF8`
`windows_866_to_windows_1251`	`WIN866`	`WIN`
`windows_874_to_utf8`	`WIN874`	`UTF8`
`euc_jis_2004_to_utf8`	`EUC_JIS_2004`	`UTF8`
`utf8_to_euc_jis_2004`	`UTF8`	`EUC_JIS_2004`
`shift_jis_2004_to_utf8`	`SHIFT_JIS_2004`	`UTF8`
`utf8_to_shift_jis_2004`	`UTF8`	`SHIFT_JIS_2004`
`euc_jis_2004_to_shift_jis_2004`	`EUC_JIS_2004`	`SHIFT_JIS_2004`
`shift_jis_2004_to_euc_jis_2004`	`SHIFT_JIS_2004`	`EUC_JIS_2004`

Notes: a. 转换名遵循一个标准的命名模式：将源编码中的所有非字母数字字符用下划线替换， 后面跟着 `_to_` ，然后后面再跟着经过相似处理的目标编码的名字。 因此这些名字可能和客户的编码名字不同。

9.4.1. 格式化

函数 `format` 生成根据格式字符串格式化了的输出，风格类似于C函数 `sprintf` 。

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`_formatstr_` 是指定结果如何格式化的格式字符串。格式字符串中的文本直接拷贝到结果中，除非已经使用了格式说明符。格式说明符在字符串中作为占位符使用， 定义后续函数参数应该格式化并且插入到结果中。每个 `_formatarg_` 参数根据这种数据类型的通常输出规则转化为文本，然后根据格式说明符格式化并且插入到结果中。

格式说明符由 `%` 字符引进，格式为

```
%[_position_][_flags_][_width_]_type_
```

组件的字段有：

`_position_` (optional)

`_n_` $格式的字符串，这里的 `_n_` 是要打印的参数的索引。索引为1表示在 `_formatstr_` 之后的第一个参数。如果省略了 `_formatstr_`， 默认使用序列中的下一个参数。

`_flags_` (optional)

附加选项，控制如何格式化格式说明符的输出。当前只支持负号(`-`)， 负号导致格式说明符的输出是左对齐的。这是没有影响的，除非指定了 `_width_` 字段。

`_width_` (optional)

声明字符数的*minimum*值用来显示格式说明符的输出。需要补充宽度时， 空白添加到输出的左侧或右侧（取决于 `-` 标志）。一个比较小的宽度不会导致输出的截断， 只是简单的忽略了。宽度可以用下列方法指定：一个正整数；一个星号(`*`) 表示使用下一个函数参数作为宽度；或一个格式为 `*``_n_` $ 的字符串表示使用第 `_n_` 个函数参数作为宽度。

如果宽度来自函数参数，那么这个参数在作为格式说明符的数值之前消耗掉。 如果宽度参数是负的，那么结果是左对齐的（就像声明了 `-` 标志一样）， 并且字段长度为 `abs` (`_width_`)。

`_type_` (required)

格式转换的类型用来产生格式说明符的输出。支持下列的类型：

- `s` 格式参数值为简单的字符串。空值作为空字符串对待。
- `I` 将参数值作为SQL标识符对待，如果需要，双写它。值为空是错误的。
- `L` 引用参数值作为SQL文字。空值用字符串 `NULL` 显示，没有引用。

除了上述的格式说明符，特殊的序列 `%%` 可以用作输出 `%` 字符。

这里有一些基本的格式转换的例子：

```
SELECT format('Hello %s', 'World');
_Result:_ <samp class="literal">Hello World</samp>

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
_Result:_ <samp class="literal">Testing one, two, three, %</samp>

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
_Result:_ <samp class="literal">INSERT INTO "Foo bar" VALUES('O''Reilly')</samp>

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', E'C:\\Program Files');
_Result:_ <samp class="literal">INSERT INTO locations VALUES(E'C:\\Program Files')</samp>
```

这里是使用 `_width_` 字段和 `-` 标志的例子：

```
SELECT format('|%10s|', 'foo');
_Result:_ <samp class="literal">|       foo|</samp>

SELECT format('|%-10s|', 'foo');
_Result:_ <samp class="literal">|foo       |</samp>

SELECT format('|%*s|', 10, 'foo');
_Result:_ <samp class="literal">|       foo|</samp>

SELECT format('|%*s|', -10, 'foo');
_Result:_ <samp class="literal">|foo       |</samp>

SELECT format('|%-*s|', 10, 'foo');
_Result:_ <samp class="literal">|foo       |</samp>

SELECT format('|%-*s|', -10, 'foo');
_Result:_ <samp class="literal">|foo       |</samp>
```

下面是使用 `_position_` 字段的例子：

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
_Result:_ <samp class="literal">Testing three, two, one</samp>

SELECT format('|%*2$s|', 'foo', 10, 'bar');
_Result:_ <samp class="literal">|       bar|</samp>

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
_Result:_ <samp class="literal">|       foo|</samp>
```

和C函数 `sprintf` 不同，PostgreSQL的 `format` 函数允许带有或不带有 `_position_` 字段的格式说明符在相同的格式字符串中混合使用。 没有 `_position_` 字段的格式说明符总是使用最后消耗参数的下一个参数。 另外， `format` 函数不要求在格式字符串中使用所有函数参数。例如：

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
_Result:_ <samp class="literal">Testing three, two, three</samp>
```

`%I` 和 `%L` 格式说明符对于安全构造动态SQL语句尤其有用。 参阅Example 40-1。

9.5. 二进制字符串函数和操作符

本节描述那些检查和操作类型为 bytea 数值的函数和操作符。

SQL定义了一些字符串函数， 在这些函数里使用关键字而不是逗号来分隔参数。 详情请见 Table 9-8。 PostgreSQL也提供了使用常用语法进行函数调用的函数的版本 (参阅Table 9-9)。

> **Note:** 本页面例子的结果在假设服务器的参数 bytea_output 设置为 escape 的基础上的（传统的PostgreSQL格式）。

Table 9-8. SQL 二进制字符串函数和操作符

函数	返回类型	描述	
string || string	bytea	字符串连接	E'\\\\Post'::bytea &#124
`octet_length(```string)	int	二进制字符串中的字节数	octet_length(E'jo\\000se
`overlay(```string placing string from int [for int])	bytea	替换子串	overlay(E'Th\\000omas'::
`position(```substring in string)	int	特定子字符串的位置	position(E'\\000om'::byt
`substring(```string [from int] [for int])	bytea	截取子串	substring(E'Th\\000omas'
trim([both]` bytes from string`)	bytea	从 string 的开头和结尾删除只包含 bytes 中字节的最长字符串	trim(E'\\000'::bytea fro

还有一些二进制字符串处理函数可以使用，在Table 9-9 列出。其中有一些是在内部使用，用于实现Table 9-8 列出的SQL标准的字符串函数。

Table 9-9. 其它二进制字符串函数

函数	返回类型	描述	例子
`btrim(```string bytea , bytes bytea)	bytea	从 string 的开头和结尾删除只包含 bytes 中字节的最长的字符串	btrim(E'\\000trim\\000'::bytea
`decode(```string text , format text)	bytea	把 string 中的文本表示解码为二进制数据。 format 的选项和 encode 相同。	decode(E'123\\000456', 'escape
`encode(```data bytea , format text)	text	把二进制数据编码为文本表现形式。支持的格式： base64 , hex , escape 。 escape 转换零字节和高位设置字节为八进制序列(\``_nnn_)和双写反斜杠。	encode(E'123\\000456'::bytea,
`get_bit(```string , offset)	int	从字符串中抽取位	get_bit(E'Th\\000omas'::bytea,
`get_byte(```string , offset)	int	从字符串中抽取字节	get_byte(E'Th\\000omas'::bytea
`length(```string)	int	二进制字符串的长度	length(E'jo\\000se'::bytea)
`md5(```string)	text	计算 string 的MD5散列值，以十六进制方式返回结果。	md5(E'Th\\000omas'::bytea)
`set_bit(```string , offset , newvalue)	bytea	设置字符串中的位	set_bit(E'Th\\000omas'::bytea,
`set_byte(```string , offset , newvalue)	bytea	设置字符串中的字节	set_byte(E'Th\\000omas'::bytea

`get_byte` 和 `set_byte` 数以二进制字符串的第一个字节为0字节。 `get_bit` 和 `set_bit` 从每个字节的右边取位； 例如位0是第一个字节的最低位，位15是第二个字节的最高位。

又见Section 9.20中的聚集函数 `string_agg` 。

9.6. 位串函数和操作符

本节描述用于检查和操作位串的函数和操作符，也就是操作类型为 `bit` 和 `bit varying` 值的函数和操作符。除了常用的比较操作符之外， 还可以使用Table 9-10里显示的操作符。 `&` , `|` , `#` 的位串操作数必须等长。在移位的时候，保留原始的位串长度(并以 0 填充)，如例子所示。

Table 9-10. 位串操作符

操作符	描述	例子	结果
`||`	连接	`B'10001' || B'011'`	`10001011`
`&`	位与	`B'10001' & B'01101'`	`00001`
`|`	位或	`B'10001'	B'01101'`	`11101`
`#`	位异或	`B'10001' # B'01101'`	`11100`
`~`	位非	`~ B'10001'`	`01110`
`<<`	位左移	`B'10001' << 3`	`01000`
`>>`	位右移	`B'10001' >> 2`	`00100`

下面的SQL标准函数除了可以用于字符串之外，也可以用于位串： `length` , `bit_length` , `octet_length` , `position` , `substring` , `overlay` 。

下面的函数用于位串和二进制字符串： `get_bit` ， `set_bit` 。当用于位串时， 这些函数位数从字符串的第一位（最左边）作为0位 。

另外，我们可以在整数和 `bit` 之间来回转换。例子：

```
44::bit(10)                    _0000101100_
44::bit(3)                     _100_
cast(-44 as bit(12))           _111111010100_
'1110'::bit(4)::integer        _14_
```

请注意，只是转换为"bit"的意思是转换成 `bit(1)` ，因此只会转换成整数的最低位。

> **Note:** 在PostgreSQL 8.0以前，把一个整数转换成 `bit(n)` 将拷贝整数的最左边的 `n` 位，而现在是拷贝最右边的 `n` 位。 还有，把一个整数转换成比整数本身长的位串，就会扩展最左边的位(非负数为 0 ， 负数为 1)。

9.7. 模式匹配

PostgreSQL提供了三种实现模式匹配的方法： 传统SQL的 `LIKE` 操作符、SQL99 新增的 `SIMILAR TO` 操作符、POSIX风格的正则表达式。 除了基本的"这个字符串匹配这个模式"操作符之外， 也可以使用函数抽取或替换匹配的子字符串并且在匹配的位置分隔字符串。

> **Tip:** 如果你的模式匹配要求比这些还多，请考虑用 Perl 或 Tcl 写一个用户定义函数。

9.7.1. `LIKE`

```
_string_ LIKE _pattern_ [ESCAPE `_escape-character_`]
_string_ NOT LIKE _pattern_ [ESCAPE `_escape-character_`]
```

如果该 `_string_` 匹配提供的 `_pattern_` ， 那么 `LIKE` 表达式返回真。和我们想像的一样，如果 `LIKE` 返回真，那么 `NOT LIKE` 表达式将返回假，反之亦然。 一个等效的表达式是 `NOT (``_string_` LIKE `_pattern_`)。

如果 `_pattern_` 不包含百分号或者下划线，那么该模式只代表它本身； 这时候 `LIKE` 的行为就像等号操作符。在 `_pattern_` 里的下划线(`_`)匹配任何单个字符；而一个百分号(`%`) 匹配零或多个任何序列。

一些例子：

```
'abc' LIKE 'abc'    _true_
'abc' LIKE 'a%'     _true_
'abc' LIKE '_b_'    _true_
'abc' LIKE 'c'      _false_
```

`LIKE` 模式匹配总是覆盖整个字符串。因此， 如果想要匹配在字符串内部任何位置的序列，该模式必须以百分号开头和结尾。

要匹配下划线或者百分号本身，在 `_pattern_` 里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜杠， 但是你可以用 `ESCAPE` 子句指定一个。要匹配逃逸字符本身， 写两个逃逸字符。

> **Note:** 如果你关闭了standard_conforming_strings选项， 那么在文本字符串常量里的任意反斜杠都需要双写。 参阅Section 4.1.2.1获取更多信息。

我们也可以通过写成 `ESCAPE ''` 的方式关闭逃逸机制， 这时，我们就不能关闭下划线和百分号的特殊含义。

关键字 `ILIKE` 可以用于替换 `LIKE` ， 令该匹配就当前的区域设置是大小写无关的。这个特性不是SQL标准， 是PostgreSQL扩展。

操作符 `~~` 等效于 `LIKE` ， 而 `~~*` 等效于 `ILIKE` 。还有 `!~~` 和 `!~~*` 操作符分别代表 `NOT LIKE` 和 `NOT ILIKE` 。 所有这些操作符都是PostgreSQL特有的。

9.7.2. `SIMILAR TO` 正则表达式

```
_string_ SIMILAR TO _pattern_ [ESCAPE `_escape-character_`]
_string_ NOT SIMILAR TO _pattern_ [ESCAPE `_escape-character_`]
```

`SIMILAR TO` 根据自己的模式是否匹配给定字符串而返回真或者假。 它和 `LIKE` 非常类似，只不过它使用 SQL 标准定义的正则表达式理解模式。 SQL 标准的正则表达式是在 `LIKE` 表示法和普通的正则表达式表示法之间古怪的交叉。

类似 `LIKE` ， `SIMILAR TO` 操作符只有在它的模式匹配整个字符串的时候才能成功； 这一点和普通的正则表达式的行为不同，在普通的正则表达式里， 模式匹配字符串的任意部分。 和 `LIKE` 类似的地方还有 `SIMILAR TO` 使用 `_` 和 `%` 分别匹配单个字符和任意字符串(这些和 POSIX 正则表达式里的 `.` 和 `.*` 兼容)。

除了这些从 `LIKE` 借用的功能之外， `SIMILAR TO` 支持下面这些从 POSIX 正则表达式借用的模式匹配元字符：

- `|` 表示选择(两个候选之一)
- `*` 表示重复前面的项零次或更多次
- `+` 表示重复前面的项一次或更多次
- `?` 表示重复前面的项零次或一次
- `{``_m_``}` 表示重复前面的项正好 `_m_` 次
- `{``_m_``,}` 表示重复前面的项 `_m_` 或更多次
- `{``_m_``,``_n_``}` 表示重复前面的项至少 `_m_` 次，最多不超过 `_n_` 次
- Parentheses `()` 把项组合成一个逻辑项
- `[...]` 声明一个字符类，只在POSIX正则表达式中

请注意点(`.`)对于 `SIMILAR TO` 来说不是元字符。

和 `LIKE` 一样，反斜杠关闭所有这些元字符的特殊含义； 当然我们也可以用 `ESCAPE` 声明另外一个逃逸字符。

一些例子：

```
'abc' SIMILAR TO 'abc'      _true_
'abc' SIMILAR TO 'a'        _false_
'abc' SIMILAR TO '%(b|d)%'  _true_
'abc' SIMILAR TO '(b|c)%'   _false_
```

带三个参数的 `substring(` *string* `from` *pattern* `for` *escape-character* `)` 函数提供了一个从字符串中抽取一个匹配 SQL 正则表达式模式的子字符串功能。 和 `SIMILAR TO` 一样，声明的模式必须匹配整个字符串， 否则函数失效并返回 NULL 。为了标识在成功的时候应该返回的模式部分， 模式必须出现后跟双引号(`"`)的两个逃逸字符。 匹配这两个标记之间的模式的字符串将被返回。

一些例子，以 `#"` 分隔返回的字符串：

```
substring('foobar' from '%#"o_b#"%' for '#')   _oob_
substring('foobar' from '#"o_b#"%' for '#')    _NULL_
```

9.7.3. POSIX 正则表达式

Table 9-11列出了所有用于 POSIX 正则表达式的操作符。

Table 9-11. 正则表达式匹配操作符

操作符	描述	例子
`~`	匹配正则表达式，大小写相关	`'thomas' ~ '.*thomas.*'`
`~*`	匹配正则表达式，大小写无关	`'thomas' ~* '.*Thomas.*'`
`!~`	不匹配正则表达式，大小写相关	`'thomas' !~ '.*Thomas.*'`
`!~*`	不匹配正则表达式，大小写无关	`'thomas' !~* '.*vadim.*'`

POSIX正则表达式提供了比 `LIKE` 和 `SIMILAR TO` 操作符更强大的模式匹配的方法。许多 Unix 工具，比如 `egrep` , `sed` , `awk` 使用类似的模式匹配语言。

正则表达式是一个字符序列，它是定义一个字符串集合(一个正则集合)的缩写。如果一个字符串是正则表达式描述的正则集合中的一员时， 我们就说这个字符串匹配该正则表达式。 和 `LIKE` 一样， 模式字符准确地匹配字符串字符，除非在正则表达式语言里有特殊字符 (不过正则表达式用的特殊字符和 `LIKE` 用的不同)。 和 `LIKE` 不一样的是，正则表达式可以匹配字符串里的任何位置， 除非该正则表达式明确地锚定在字符串的开头或者结尾。

一些例子：

```
'abc' ~ 'abc'    _true_
'abc' ~ '^a'     _true_
'abc' ~ '(b|d)'  _true_
'abc' ~ '^(b|c)' _false_
```

POSIX模式语言将在下面详细描述。

带两个参数的 `substring(` *string* `from` *pattern* `)` 函数提供了从字符串中抽取一个匹配 POSIX 正则表达式模式的子字符串的方法。 如果没有匹配它返回 NULL， 否则就是文本中匹配模式的那部分。 但是如果该模式包含任何圆括弧，那么将返回匹配第一对子表达式(对应第一个左圆括弧的)的文本。 如果你想在表达式里使用圆括弧而又不想导致这个例外， 那么你可以在整个表达式外边放上一对圆括弧。如果你需要在想抽取的子表达式前有圆括弧， 参阅描述的非捕获性圆括弧。

一些例子：

```
substring('foobar' from 'o.b')     _oob_
substring('foobar' from 'o(.)b')   _o_
```

`regexp_replace` (`_source_` , `_pattern_` , `_replacement_` [, `_flags_`])函数提供了将匹配 POSIX 正则表达式模式的子字符串替换为新文本的功能。如果没有匹配 pattern 的子字符串， 那么返回不加修改的 `_source_` 字符串。如果有匹配， 则返回的 `_source_` 字符串里面的对应子字符串将被 `_replacement_` 字符串替换掉。 `_replacement_` 字符串可以包含 `\``_n_` ，这里的 `_n_` 是 1 到 9 ，表明源字符串中匹配第 `_n_` 个圆括弧子表达式的部分将插入在该位置，并且它可以包含 `\&` 表示应该插入匹配整个模式的字符串。如果你需要放一个文本反斜杠在替换文本里， 那么写 `\\` 。可选的 `_flags_` 参数包含零个或多个改变函数行为的单字母标记。 `i` 表示进行大小写无关的匹配， `g` 表示替换每一个匹配的子字符串而不仅仅是第一个。其他支持的标记在 Table 9-19中描述。

一些例子：

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   _fooXbaz_
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   _fooXX_
regexp_replace('foobarbaz', 'b(..)', E'X\\1Y', 'g')
                                   _fooXarYXazY_
```

`regexp_matches` (`_string_` , `_pattern_` [, `_flags_`])函数返回一个从匹配POSIX正则表达式模式中获取的所有子串结果的text数组。 这个函数可以返回零行，一行，或者多行（参阅下面的 `g` 标记）。如果 `_pattern_` 没有匹配，则函数返回零行。如果模式包含没有括号的子表达式，则每行返回的是单元素的文本数组， 其中包含的子串相匹配整个模式。如果模式包含有括号的子表达式，函数返回一个文本数组，它的第 `_n_` 个元素是子串匹配模式括号子表达式内的第 `_n_` 个元素。 （不计"非捕获"的括号；详细信息参阅下面）。参数 `_flags_` 是一个可选的text字符串， 含有0或者更多单字母标记来改变函数行为。标记 `g` 导致查找字符串中的每个匹配，而不仅是第一个， 每个匹配返回一行，其它支持的标记在Table 9-19里描述。

一些例子：

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
 regexp_matches
----------------
 {bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
 regexp_matches
----------------
 {barbeque}
(1 row)
```

使用select子句，可能强制 `regexp_matches()` 总是返回一行；当你想要返回 `SELECT` 目标列表中的所有行，甚至没有匹配的情况下，是有特别有用的。

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

`regexp_split_to_table` (`_string_` , `_pattern_` [, `_flags_`])函数使用POSIX正则表达式模式作为分隔符，分隔字符串。 如果没有匹配 `_pattern_` ，函数将返回 `_string_` 。如果有至少一个匹配， 每个匹配返回从最后一个匹配结束（或者字符串的开头）到匹配开始的文本。当没有更多的匹配， 返回最后一个匹配的结束到字符串的结束的文本。 `_flags_` 参数是一个可选text字符串， 含有0或者更多单字母标记来改变函数行为。 `regexp_split_to_table` 支持的标记在Table 9-19里描述。

除了 `regexp_split_to_array` 返回结果为text数组， `regexp_split_to_array` 函数行为与 `regexp_split_to_table` 相同，使用语法 `regexp_split_to_array` (`_string_` , `_pattern_` [, `_flags_`])。 参数与 `regexp_split_to_table` 相同。

一些例子：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', E'\\
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', E'\\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

作为最后一个例子表明，发生在字符串的开始或结束或紧接前一个的匹配，regexp分隔函数忽略零长度匹配， 这样实现 `regexp_matches` 严格上来说是违背了的正则表达式匹配的定义，但在实际使用中， 通常是最便利的的行为。如Perl等软件系统，使用了类似的定义。

9.7.3.1. 正则表达式细节

PostgreSQL的正则表达式使用 Henry Spencer 写的一个包来实现。 下面的大部分描述都是从他的手册页里逐字拷贝过来的。

正则表达式(REs)，在POSIX 1003.2中定义， 它有两种形式：扩展 RE或ERE (基本上就是在 `egrep` 里的那些)，基本_RE 或*BRE(*基本上就是在 `ed` 里的那些*)*。*PostgreSQL* 两种形式都实现了，并且还做了一些 *POSIX* 里面没有的， 但是因为在类似 *Perl* 或者 *Tcl* 这样的语言中得到广泛应用的一些扩展。 使用了那些非 *POSIX* 扩展的正则表达式叫高级_RE 或ARE。 ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容(以及更多的限制)。 我们首先描述 ARE 和 ERE 形式，描述那些只适用于 ARE 的特性，然后描述与 BRE 的区别是什么。

> **Note:** PostgreSQL总是初始化一个遵循ARE规则的正则表达式。然而， 更多限制的ERE或BRE规则可以通过在RE模式前放置一个*embedded option*来选择， 描述在Section 9.7.3.4。这对于期望完全兼容POSIX 1003.2规则的应用程序是有用的。

一个正则表达式定义为一个或多个分支， 由 `|` 分隔。它匹配其中任何一个分支的东西。

一个分支是零个或多个有修饰的原子或约束连接而成。 一个原子匹配第一个，后面的原子匹配第二个，以此类推；一个空分支匹配空字符串。

一个有修饰的原子是一个原子，后面可能跟着一个量词。 没有量词的时候，它匹配一个原子，有量词的时候，它可以匹配若干个原子。 原子可以是在Table 9-12里面显示的任何可能。 可能的量词和他们的含义在Table 9-13里显示。

一个约束匹配一个空字符串，但只是在满足特定条件下才匹配。 约束可以在能够使用原子的地方使用，只是它不能跟着量词。 最简单的原子在Table 9-14里显示；更多的约束稍后描述。

Table 9-12. 正则表达式原子

原子	描述
`(``_re_``)`	(`_re_` 是任意正则表达式)匹配一个对 `_re_` 的匹配，有可报告的匹配信息。
`(?:``_re_``)`	同上，但是匹配不会被报告(一个"非捕获"圆括弧)，只在 ARE 中有。
`.`	匹配任意单个字符
`[``_chars_``]`	一个方括弧表达式，匹配任意的 `_字符_` (参阅Section 9.7.3.2获取更多细节)
`\``_k_`	(`_k_` 是非字母数字字符)匹配一个当作普通字符看待的特定字符， 比如 `\\` 匹配一个反斜杠。
`\``_c_`	`_c_` 是一个字母数字(可能跟着其它字符)，它是一个逃逸， 参阅Section 9.7.3.3。仅存在于 ARE 中；在 ERE 和 BRE 中， 它匹配 `_c_` 。
`{`	如果后面跟着一个非数字字符，那么就匹配左花括弧 `{` ；如果跟着一个数字，那么它是 `_范围_` 的开始(见下面)
`_x_`	这里的 `_x_` 是一个没有其它特征的单个字符，则匹配该字符

RE不能以(`\`)结尾。

> **Note:** 如果关闭了standard_conforming_strings， 任何文本字符串常量中的反斜杠都需要双写。参阅Section 4.1.2.1 获取更多信息。

Table 9-13. 正则表达式量词

量词	匹配
`*`	一个匹配 0 或者更多个原子的序列
`+`	一个匹配 1 或者更多个原子的序列
`?`	一个匹配 0 或者 1个原子的序列
`{``_m_``}`	一个正好匹配 `_m_` 个原子的序列
`{``_m_``,}`	一个匹配 `_m_` 个或者更多原子的序列
`{``_m_``,``_n_``}`	一个匹配 `_m_` 到 `_n_` 个(包含两端)原子的序列； `_m_` 不能比 `_n_` 大
`*?`	`*` 的非贪婪模式
`+?`	`+` 的非贪婪模式
`??`	`?` 的非贪婪模式
`{``_m_``}?`	`{``_m_``}` 的非贪婪模式
`{``_m_``,}?`	`{``_m_``,}` 的非贪婪模式
`{``_m_``,``_n_``}?`	`{``_m_``,``_n_``}` 的非贪婪模式

`{``_..._``}` 的形式被称作范围。 一个范围内的数字 `_m_` 和 `_n_` 都是无符号十进制整数， 允许的数值从 0 到 255 (闭区间)。

非贪婪的量词(只在 ARE 中可用)匹配对应的正常(贪婪)模式， 区别是它寻找最少的匹配，而不是最多的匹配。参阅Section 9.7.3.5 获取细节。

> **Note:** 一个量词不能紧跟在另外一个量词后面，例如， `**` 是非法的。 量词不能是表达式或者子表达式的开头，也不能跟在 `^` 或 `|` 后面。

Table 9-14. 正则表达式约束

约束	描述
`^`	匹配字符串的开头
`$`	匹配字符串的结尾
`(?=``_re_``)`	正前瞻匹配任何匹配 `_re_` 的子字符串起始点(只在 ARE 中有)
`(?!``_re_``)`	负前瞻匹配任何不匹配 `_re_` 的子字符串起始点(只在 ARE 中有)

前瞻约束不能包含后引用(参阅Section 9.7.3.3)， 并且在其中的所有圆括弧都被认为是不捕获的。

9.7.3.2. 方括弧表达式

方括弧表达式是一个包围在 `[]` 里的字符列表。 它通常匹配任意单个列表中的字符(又见下文)。如果列表以 `^` 开头， 它匹配任意单个(又见下文)不在该列表中的字符。如果该列表中两个字符用 `-` 隔开，那它就是那两个字符(包括在内)之间的所有字符范围的缩写， 比如，在ASCII里 `[0-9]` 包含任何十进制数字。 两个范围共享一个终点是非法的，比如 `a-c-e` 。这个范围与字符集关系密切， 可移植的程序不应该依靠它们。

想在列表中包含文本 `]` ，可以让它做列表的首字符(如果用到了， 跟在 `^` 后面)。想在列表中包含文本 `-` ， 可以让它做列表的首字符或者末字符，或者一个范围的第二个终点。 想在列表中把文本 `-` 当做范围的起点，把它用 `[.` 和 `.]` 包围起来，这样它就成为一个集合元素(见下文)。 除了这些字符本身，和一些用 `[` 的组合(见下段)， 以及逃逸(只在 ARE 中有效)以外，所有其它特殊字符在方括弧表达式里都失去它们的特殊含义。 特别是，在 ERE 和 BRE 规则下 `\` 不是特殊的，但在 ARE 里， 它是特殊的(还是引入一个逃逸)。

在一个方括弧表达式里，一个集合元素(一个字符、一个当做一个字符的多字符序列、 或者一个表示上面两种情况的集合序列)包含在 `[.` 和 `.]` 里面的时候表示该集合元素的字符序列。该序列是该方括弧列表的一个元素。 这允许一个包含多字符集合元素的方括弧表达式就可以匹配多于一个字符，比如， 如果集合序列包含一个 `ch` 集合元素，那么 `[[.ch.]]*c` 匹配 `chchcc` 的头五个字符。译注：其实把 [. 和 .] 括起来的整体当一个字符看就行了。

> **Note:** PostgreSQL目前不支持多字符集合元素。这些信息描述了将来可能有的行为。

在方括弧表达式里，在 `[=` 和 `=]` 里包围的集合元素是一个等效表，代表等于这里所有集合元素的字符序列， 包括它本身(如果没有其它等效集合元素，那么就好像封装元素是 `[.` 和 `.]`)。比如，如果 `o` 和 `^` 是一个等效表的成员，那么 `[[=o=]]` , `[[=^=]]` , `[o^]` 都是同义的。一个等效表不能是一个范围的端点。

在方括弧表达式里，在 `[:` 和 `:]` 里面封装的字符表名字代表属于该表的所有字符的列表。标准的字符表名字是： `alnum` , `alpha` , `blank` , `cntrl` , `digit` , `graph` , `lower` , `print` , `punct` , `space` , `upper` , `xdigit` 。 它们代表在ctype 里定义的字符表。本地化设置可能会提供其它的表。字符表不能用做一个范围的端点。

在方括弧表达式里有两个特例：方括弧表达式 `[[:<:]]` 和 `[[:>:]]` 是约束，分别匹配一个单词开头和结束的空串。 单词定义为一个单词字符序列，前面和后面都没有其它单词字符。 单词字符是一个 `alnum` 字符(和ctype 里定义的一样)或者一个下划线。这是一个扩展，兼容POSIX 1003.2 ， 但那里面并没有说明，而且在准备移植到其它系统里去的软件里一定要小心使用。 通常下面描述的约束逃逸更好些；他们并非更标准，但是更容易输入。

9.7.3.3. 正则表达式逃逸

逃逸是以 `\` 开头，后面跟着一个字母数字字符的特殊序列。 逃逸有好几种变体：字符项、表缩写、约束逃逸、后引用。在 ARE 里，如果一个 `\` 后面跟着一个字母数字，但是并未组成一个合法的逃逸，那么它是非法的。在 ERE 里则没有逃逸： 在方括弧表达式之外，一个跟着

字母数字字符的 `\` 只是表示该字符是一个普通的字符， 而在一个方括弧表达式里， `\` 是一个普通的字符(后者实际上是 ERE 和 ARE 之间的不兼容)。

字符项逃逸用于方便我们声明正则表达式里那些不可打印的字符。 它们在Table 9-15里列出。

类缩写逃逸用来提供一些常用的字符类缩写。 他们在Table 9-16里列出。

约束逃逸是一个约束，如果满足特定的条件，它匹配该空字符串，以逃逸形式写出。 它们在Table 9-17里列出。

后引用(`` \``_n_ ``)匹配数字 `_n_` 指定的前面的圆括弧子表达式匹配的同一个字符串(参阅Table 9-18)。 比如， `([bc])\1` 匹配 `bb` 或 `cc` 但是不匹配 `bc` 或 `cb` 。 正则表达式里的子表达式必须完全在后引用前面。子表达式以它的括号的顺序排序。 非捕获圆括弧并不定义子表达式。

> **Note:** 请注意，如果把模式当作一个 SQL 字符串常量输入，那么逃逸前导的 `\` 需要双倍地写：
>
> ```
> '123' ~ E'^\\d{3}' _true_
> ```

Table 9-15. 正则表达式字符项逃逸

逃逸	描述
`\a`	警笛(铃声)字符，和 C 里一样
`\b`	退格，和 C 里一样
`\B`	`\` 的同义词，用于减少反斜杠加倍的需要
`\c``_X_`	(这里 `_X_` 是任意字符)字符的低 5 位和 `_X_` 里的相同，其它位都是 0
`\e`	集合序列名字是 `ESC` 的字符，如果不是， 则是八进制值为 033 的字符
`\f`	进纸，和 C 里一样
`\n`	新行，和 C 里一样
`\r`	回车，和 C 里一样
`\t`	水平制表符，和 C 里一样
`\u``_wxyz_`	(这里的 `_wxyz_` 是恰好四位十六进制位)本机字节序的 UTF-16 字符 `U+``_wxyz_`
`\U``_stuvwxyz_`	(这里的 `_stuvwxyz_` 是恰好八位十六进制位) 为假想中的 Unicode 32 位扩展保留的
`\v`	垂直制表符，和 C 里一样
`\x``_hhh_`	(这里的 `_hhh_` 是一个十六进制序列)十六进制值为 `0x``_hhh_` 的字符(不管用了几个十六进制位， 都是一个字符)
`\0`	值为 `0` 的字符 (null 字节)
`\``_xy_`	(这里的 `_xy_` 是恰好两个八进制位， 并且不是一个后引用)八进制值为 `0``_xy_` 的字符
`\``_xyz_`	(这里的 `_xyz_` 是恰好三位八进制位， 并且不是一个后引用)八进制值为 `0``_xyz_` 的字符

十六进制位是 `0` - `9`，`a` - `f`，`A` - `F`。八进制位是 `0` - `7`。

字符项逃逸总是被当作普通字符。比如， `\135` 是 ASCII 中的 `]`， 但 `\135` 并不终止一个方括弧表达式。

Table 9-16. 正则表达式类缩写逃逸

逃逸	描述
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (注意，这里是包含下划线的)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (注意，这里是包含下划线的)

在方括弧表达式里，\d，\s，\w 会失去他们的外层方括弧，而 \D，\S，\W 是非法的。比如 [a-c\d] 等效于 [a-c[:digit:]]。同样 [a-c\D] 原来等效于 [a-c^[:digit:]] 的，也是非法的。

Table 9-17. 正则表达式约束逃逸

逃逸	描述
\A	只匹配字符串开头(参阅Section 9.7.3.5 获取它和 ^ 区别的信息)
\m	只匹配一个词的开头
\M	只匹配一个词的结尾
\y	只匹配一个词的开头或者结尾
\Y	只匹配那些既不是词的开头也不是词的结尾的点
\Z	只匹配一个字符串的结尾(参阅Section 9.7.3.5 获取它和 $ 区别的信息)

一个词的定义是上面 [[:<:]] 和 [[:>:]] 的声明。在方括弧表达式里，约束逃逸是非法的。

Table 9-18. 正则表达式后引用

逃逸	描述
\``_m_	(这里的 _m_ 是一个非零十进制位) 一个指向第 _m_ 个子表达式的后引用
\``_mnn_	(这里的 _m_ 是一个非零十进制位， _nn_ 是更多的十进制位，并且十进制数值 _mnn_ 不能大于到这个位置为止的闭合捕获圆括弧的个数)一个指向第 _mnn_ 个子表达式的后引用

> **Note:** 在八进制字符项逃逸和后引用之间有一个继承的歧义存在，这个歧义是通过跟着的启发分析解决的， 像上面描述的那样。前导零总是表示这是一个八进制逃逸。而单个非零数字， 如果没有跟着任何其它数字，那么总是认为是后引用。 一个多数据位的非零开头的序列也认为是后引用(只要它在合适的子表达式后面， 也就是说，数值在后引用的合法范围内)，否则就认为是一个八进制。

9.7.3.4. 正则表达式元语法

除了上面描述的主要语法之外，还有几种特殊形式和杂项语法。

正则表达式可以以两个特殊的指示器前缀之一开始： 如果一个正则表达式以 `***:` 开头，那么剩下的正则表达式都被当作 ARE 。 （这在PostgreSQL中通常没有影响，因为正则表达式被假设为ARE； 但是如果ERE或BRE模式被 `_flags_` 参数指定为正则表达式函数时是有影响的。） 如果一个的正则表达式以 `***=` 开头，那么剩下的正则表达式被当作一个文本串， 所有的字符都被认为是一个普通字符。

一个 ARE 可以以嵌入选项开头：一个 `(?``_xyz_``)` 序列(这里的 `_xyz_` 是一个或多个字母字符)声明影响剩余正则表达式的选项。 这些选项覆盖任何前面判断的选项—它们可以重写正则表达式操作符隐含的大小写敏感性， 或者正则表达式函数的 `_flags_` 参数。可用的选项字母在 Table 9-19显示。请注意， 正则表达式函数的 `_flags_` 参数使用相同的选项字母。

Table 9-19. ARE 嵌入选项字母

选项	描述
`b`	剩余的正则表达式是 BRE
`c`	大小写敏感匹配(覆盖操作符类型)
`e`	剩余的正则表达式是 ERE
`i`	大小写不敏感匹配(参阅Section 9.7.3.5)(覆盖操作符类型)
`m`	`n` 的历史同义词
`n`	新行敏感匹配(参阅Section 9.7.3.5)
`p`	部分新行敏感匹配(参阅Section 9.7.3.5)
`q`	重置正则表达式为一个文本("引起")字符串，所有都是普通字符。
`s`	非新行敏感匹配(缺省)
`t`	紧语法(缺省， 见下文)
`w`	反转部分新行敏感("怪异")匹配(参阅Section 9.7.3.5)
`x`	扩展的语法(见下文)

嵌入的选项在终止其序列的 `)` 发生作用。他们只在 ARE 的开始处起作用(如果有， 则在任何 `***:` 指示器后面)。

除了通常的(紧)正则表达式语法(这种情况下所有字符都重要)， 还有一种扩展语法，可以通过声明嵌入的 `x` 选项获得。 在扩展语法里，正则表达式中的空白字符被忽略，就像那些在 `#` 和新行之间的字符一样（或正则表达式的结尾）。 这样就允许我们给一个复杂的正则表达式分段和注释。不过这个基本规则上有三种例外：

- 前置了 `\` 的空白字符或者 `#` 保留
- 方括弧里的空白或者 `#` 保留

- 在多字符符号里面不能出现空白和注释，比如 `(?:`

在这里，空白是空格、水平制表符、新行、和任何属于 `_space_` (空白)字符表的字符。

最后，在 ARE 里，方括弧表达式外面，序列 `(?#``_ttt_``)` (这里的 `_ttt_` 是任意不包含 `)` 的文本)是一个注释，完全被忽略。 同样，这样的东西是不允许出现在多字符符号的字符中间的，比如 `(?:`。 这样的注释是比有用的机制的更久远的历史造成的，他们的用法已经废弃了； 我们应该使用扩展语法代替他。

如果声明了一个初始化的 `***=` 指示器，那么所有这些元语法扩展都 不能使用，因为这样表示把用户输入当作一个文本字符串而不是正则表达式对待。

9.7.3.5. 正则表达式匹配规则

在正则表达式可以匹配给出的字符串中多于一个子字符串的情况下， 正则表达式匹配字符串中最靠前的那个子字符串。 如果正则表达式可以匹配在那个位置开始的多个子字符串，要么是取最长的子字符串， 要么是最短的，具体哪种，取决于正则表达式是贪婪的还是 非贪婪的。

一个正则表达式是否贪婪取决于下面规则：

- 大多数原子，以及所有约束，都没有贪婪属性(因为它们毕竟无法匹配个数变化的文本)。
- 在一个正则表达式周围加上圆括弧并不会改变其贪婪性。
- 一个带一个固定重复次数的量词(`{``_m_``}` 或 `{``_m_``}?`) 量化的原子和原子自身有着同样的贪婪性(可能是没有)。
- 一个带其它普通的量词(包括 `{``_m_``,``_n_``}` 中 `_m_` 等于 `_n_` 的情况)量化的原子是贪婪的(首选最长匹配)。
- 一个带非贪婪量词(包括 `{``_m_``,``_n_``}?` 中 `_m_` 等于 `_n_` 的情况)量化原子是非贪婪的(首选最短匹配)。
- 一个分支(也就是一个没有顶级 `|` 操作的正则表达式) 和它里面的第一个有贪婪属性的量化原子有着同样的贪婪性。
- 一个由 `|` 操作符连接起来的两个或者更多分支组成的正则表达式总是贪婪的。

上面的规则所描述的贪婪属性不仅仅适用于独立的量化原子， 而且也适用于包含量化原子的分支和整个正则表达式。这里的意思是， 匹配是按照分支或者整个正则表达式作为一个整体匹配最长或者最短的子字符串的可能。一旦整个匹配的长度确定， 那么匹配任意子表达式的部分就基于该子表达式的贪婪属性进行判断， 在正则表达式里面靠前的子表达式的优先级高于靠后的子表达式。

一个表达这些的例子：

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
_Result:_ <samp class="literal">123</samp>
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
_Result:_ <samp class="literal">1</samp>
```

在第一个例子里，正则表达式作为整体是贪婪的，因为 `Y*` 是贪婪的。 它可以匹配从 `Y` 开始的东西，并且它匹配从这个位置开始的最长的字符串， 也就是 `Y123` 。输出是这里的圆括弧包围的部分，或者说是 `123` 。 在第二个例子里，正则表达式总体上是一个非贪婪的正则表达式，因为 `Y*?` 是非贪婪的。它可以匹配从 `Y` 开始的最短的子字符串，也就是说 `Y1` 。 子表达式 `[0-9]{1,3}` 是贪婪的，但是它不能修改总体匹配长度的决定；因此它被迫只匹配 `1` 。

简单说，如果一个正则表达式同时包含贪婪和非贪婪的子表达式， 那么总匹配长度要么是最长可能，要么是最短可能，取决于给整个正则表达式赋予的贪婪属性。 给子表达式赋予的贪婪属性只影响在这个匹配里，各个子表达式之间相互允许 "吃进"的多少。

量词 `{1,1}` 和 `{1,1}?` 可以分别用于在一个子表达式或者整个正则表达式上强制贪婪或者非贪婪。

匹配长度是以字符衡量的，而不是集合的元素。一个空字符串会被认为比什么都不匹配长。比如： `bb*` 匹配 `abbbc` 的中间三个字符； `(week|wee)(night|knights)` 匹配 `weeknights` 的所有十个字符；而 `(.*).*` 匹配 `abc` 的时候，圆括弧包围的子表达式匹配所有三个字符；而如果用 `(a*)*` 匹配 `bc` ，那么正则表达式和圆括弧子表达式都匹配空字符串。

如果声明了大小写无关的匹配，那么效果就好像把所有字母上的大小写区别取消了一样。 如果一个存在大小写差别的字母以一个普通字符的形式出现在方括弧表达式外面， 那么它实际上被转换成一个包含大小写的方括弧表达式，也就是说， `x` 变成 `[xX]` 。如果它出现在一个方括弧表达式里面， 那么它的所有大小写的同族都被加入方括弧表达式中，也就是说， `[x]` 变成 `[xX]` 而 `[^x]` 变成 `[^xX]` 。

如果声明了新行敏感匹配， `.` 和使用 `^` 的方括弧表达式将永远不会匹配新行字符(这样，匹配就绝对不会跨新行， 除非正则表达式明确地安排了这样的情况)并且 `^` 和 `$` 除了分别匹配字符串开头和结尾之外，还将分别匹配新行后面和前面的空字符串。 但是 ARE 逃逸 `\A` 和 `\Z` 仍然只匹配字符串的开头和结尾。

如果声明了部分新行敏感匹配，那么它影响 `.` 和方括弧表达式， 这个时候和新行敏感匹配一样，但是不影响 `^` 和 `$` 。

如果声明了反转部分新行敏感匹配，那么它影响 `^` 和 `$` ， 作用和新行敏感匹配里一样，但是不影响 `.` 和方括弧表达式。 这个没什么太多用途，只是为了对称提供的。

9.7.3.6. 限制和兼容性

在这个实现里，对正则表达式的长度没有特别的限制，但是， 那些希望能够有很好移植行的程序应该避免写超过 256 字节的正则表达式， 因为 POSIX 兼容的实现可以拒绝接受这样的正则表达式。

ARE 实际上和 POSIX ERE 不兼容的唯一的特性是在方括弧表达式里 `\` 并不失去它特殊的含义。所有其它 ARE 特性都使用在 POSIX ERE 里面是非法或者是未定义、 未声明效果的语法；指示器的 `***` 就是在 POSIX 的 BRE 和 ERE 之外的语法。

许多 ARE 扩展都是从 Perl 那里借来的，但是有些我做了修改，清理了一下， 以及一些 Perl 里没有出现的扩展。要注意的不兼容包括 `\b`，`\B`， 对结尾的新行缺乏特别的处理，对那些新行敏感匹配的附加的补齐方括弧表达式， 在前瞻约束里对圆括弧和方括弧引用的限制，以及最长/最短匹配(而不是第一匹配)语义。

PostgreSQL 7.4 之前的版本里的 ARE 和 ERE 存在两个非常显著的不兼容：

- 在 ARE 里，后面跟着一个字母数字的 `\` 要么是一个逃逸，要么是错误， 但是在以前的版本里，它只是写那个字母数字的另外一种方法。这个应该不是什么问题， 因为在以前的版本里没有什么原因让我们写这样的序列。
- 在 ARE 里， `\` 在 `[]` 里还是一个特殊字符， 因此在方括弧表达式里的一个文本 `\` 必须写成 `\\`。

9.7.3.7. 基本正则表达式

BRE 在几个方面和 ERE 不太一样。在BRE里， `|`，`+`，`?` 都是普通字符，它们没有等效的功能替换。范围的分隔符是 `\{` 和 `\}`， 因为 `{` 和 `}` 本身是普通字符。嵌套的子表达式的圆括弧是 `\(` 和 `\)`，因为 `(` 和 `)` 自身是普通字符。 除非在正则表达式开头或者是圆括弧封装的子表达式开头， `^` 都是普通字符， 除非在正则表达式结尾或者是圆括弧封装的子表达式的结尾， `$` 是一个普通字符， 而如果 `*` 出现在正则表达式开头或者是圆括弧封装的子表达式开头 (前面可能有 `^`)，那么它是个普通字符。最后，可以用单数字的后引用， 以及 `\<` 和 `\>` 分别是 `[[:<:]]` 和 `[[:>:]]` 的同义词；在BRE里没有其它的逃逸。

9.8. 数据类型格式化函数

PostgreSQL格式化函数提供一套有效的工具用于把各 种数据类型转换成格式化的字符串以及反过来从格式化的字符串转换成指定的数据类型。 Table 9-20列出了这些函数。 这些函数都遵循一个公共的调用习惯：第一个参数是待格式化的值， 而第二个是定义输出或输入格式的模板。

单参数 `to_timestamp` 函数也可以使用；它接受一个 `double precision` 参数，并且从Unix纪元（秒自1970-01-01 00:00:00+00）转换为 `timestamp with time zone` 类型。（ `Integer` 类型的Unix纪元隐含的转换为 `double precision` 类型。）

Table 9-20. 格式化函数

函数	返回类型	描述	
`to_char(```timestamp , text)	`text`	把时间戳转换成字符串	`to_char(cur`
`to_char(```interval , text)	`text`	把时间间隔转为字符串	`to_char(int`
`to_char(```int , text)	`text`	把整数转换成字符串	`to_char(125`
to_char`( double precision , text`)	`text`	把实数/双精度数转换成字符串	`to_char(125`
`to_char(```numeric , text)	`text`	把数字转换成字符串	`to_char(-12`
`to_date(```text , text)	`date`	把字符串转换	`to_date('05`

		成日期	
`to_number(```text`, `text`)	`numeric`	把字符串转换成数字	`to_number('
`to_timestamp(```text`, `text`)	`timestamp with time zone`	把字符串转换成时间戳	`to_timestam
`to_timestamp(```double precision`)	`timestamp with time zone`	把Unix纪元转换成时间戳	`to_timestam

在 `to_char` 输出模板字符串里，该函数族可以识别一些特定的模式， 并且把给定的数值正确地格式化成相应的数据。任何不属于模板模式的文本都简单地逐字拷贝。 同样，在一个输入模板字符串里(对其他函数)，模板模式标识数值由输入数据字符串提供。

Table 9-21显示了可以用于格式化日期和时间值的模版。

Table 9-21. 用于日期/时间格式化的模式

模式	描述
`HH`	一天的小时数(01-12)
`HH12`	一天的小时数(01-12)
`HH24`	一天的小时数(00-23)
`MI`	分钟(00-59)
`SS`	秒(00-59)
`MS`	毫秒(000-999)
`US`	微秒(000000-999999)
`SSSS`	午夜后的秒(0-86399)
`AM`, `am`, `PM` 或 `pm`	正午指示器(没有周期)
`A.M.`, `a.m.`, `P.M.` 或 `p.m.`	正午指示器(有周期)
`Y,YYY`	带逗号的年(4 和更多位)
`YYYY`	年(4 和更多位)

`YYY`	年的后三位
`YY`	年的后两位
`Y`	年的最后一位
`IYYY`	ISO年(4 位或更多位)
`IYY`	ISO年的最后三位
`IY`	ISO年的最后两位
`I`	ISO年的最后一位
`BC` , `bc` , `AD` 或 `ad`	纪元标识(没有周期)
`B.C.` , `b.c.` , `A.D.` 或 `a.d.`	纪元标识(有周期)
`MONTH`	全长大写月份名(空白填充为 9 字符)
`Month`	全长首字母大写月份名(空白填充为 9 字符)
`month`	全长小写月份名(空白填充为 9 字符)
`MON`	大写缩写月份名(英文3 字符，本地化的长度不同)
`Mon`	首字母大写缩写月份名(英文3 字符，本地化的长度不同)
`mon`	小写缩写月份名(英文3 字符，本地化的长度不同)
`MM`	月份数(01-12)
`DAY`	全长大写日期名(空白填充为 9 字符)
`Day`	全长首字母大写日期名(空白填充为 9 字符)
`day`	全长小写日期名(空白填充为 9 字符)
`DY`	缩写大写日期名(英文3 字符，本地化长度不同)
`Dy`	缩写首字母大写日期名(英文3 字符，本地化长度不同)
`dy`	缩写小写日期名(英文3 字符，本地化长度不同)
`DDD`	一年里的日(001-366)
`IDDD`	ISO一年里的日（001-371；年的第一天是第一个ISO周的星期一）
`DD`	一个月里的日(01-31)
`D`	一周里的日，星期日（ `1` ）到星期六（ `7` ）
`ID`	ISO一周里的日，星期一（ `1` ）到星期日（ `7` ）
`W`	一个月里的周数(1-5)(第一周从该月第一天开始)
`WW`	一年里的周数(1-53)(第一周从该年的第一天开始)
`IW`	ISO一年里的周数(01-53；第一个星期四在第一周里)
`CC`	世纪(2 位)(20 世纪从 2001-01-01 开始)

`J`	儒略日(自公元前 4714 年 11 月 24日午夜来的天数)
`Q`	季度（`to_date` 和 `to_timestamp` 忽略此项）
`RM`	罗马数字的月份(I-XII ；I=January)(大写)
`rm`	罗马数字的月份(i-xii ；i=January)(小写)
`TZ`	时区名(大写)
`tz`	时区名(小写)

有一些修饰词可以应用于模板来修改它们的行为。比如，`FMMonth` 就是带着 `FM` 前缀的 `Month` 模式。 Table 9-22 显示了用于日期/时间格式化的修饰词模式。

Table 9-22. 日期/时间格式化的模板模式修饰词

修饰词	描述	例子
`FM` 前缀	填充模式(抑制填充空白和尾随零)	`FMMonth`
`TH` 后缀	大写顺序数后缀	`DDTH`，例如，`12TH`
`th` 后缀	小写顺序数后缀	`DDth`，例如，`12th`
`FX` 前缀	固定格式全局选项(见用法须知)	`FX Month DD Day`
`TM` 前缀	翻译模式(基于lc_time显示本地化的日期和月份名)	`TMMonth`
`SP` 后缀	拼写模式(未实现)	`DDSP`

日期/时间格式化的用法须知：

- `FM` suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width. In PostgreSQL, `FM` modifies only the next specification, while in Oracle `FM` affects all subsequent specifications, and repeated `FM` modifiers toggle fill mode on and off.

  `FM` 抑制前导的零或尾随的空白，如果没有使用它的话， 会在输出中增加这些填充最终把输出变成固定宽度的模式。 在PostgreSQL中， `FM` 只修改下一个规范， 而在Oracle中 `FM` 影响所有随后的规范， 并且重复 `FM` 修饰符填充模式开关打开或关闭。

- `TM` 不包含结尾空白。

- 如果没有使用 `FX`， `to_timestamp` 和 `to_date` 在转换字符串的时候忽略多个空白。比如 `to_timestamp('2000 JUN', 'YYYY MON')` 是正确的，但是 `to_timestamp('2000 JUN', 'FXYYYY MON')` 会返回一个错误，因为 `to_timestamp` 只预料会有一个空白。 `FX` 必须做为模板里的第一个项声明。

- 在 `to_char` 模板里可以有普通文本，并且它们会被逐字输出。 你可以把一个字符串放到双引号里强迫它解释成一个文本，即使它里面包含模式关键字也如此。 比如 `'"Hello Year "YYYY'` 中的 `YYYY` 将被年份数据代替， 但是 `Year` 里单独的 `Y` 会。

在 `to_date` ， `to_number` 和 `to_timestamp` 里，加双引号的字符串忽略包含在字符串中的输入字符的数量， 例如， `"XX"` 忽略两个输入字符。

- 如果你想在输出里有双引号，那么你必须在它们前面放反斜杠，比如 `'\"YYYY Month\"'` 。
- 如果年份的格式规范少于四个字节，例如 `YYY` ，并且提供的年份少于四个字节， 那么年份将调整为接近于2020，例如 `95` 成为1995。
- 如果你使用的年份长于4位字符，那么用 `YYYY` 从字符串向 `timestamp` 或 `date` 做转换时要受到限制。你必须在 `YYYY` 后面使用一些非数字字符或者模板，否则年份总是解释为 4 位数字。比如对于 20000 年： `to_date('200001131', 'YYYYMMDD')` 将会被解释成一个 4 位数字的年份， 最好在年后面使用一个非数字的分隔符，像 `to_date('20000-1131', 'YYYY-MMDD')` 或 `to_date('20000Nov31', 'YYYYMonDD')` 。
- 在从字符串向 `timestamp` 或 `date` 转换的时候，如果有 `YYY` , `YYYY` 或 `Y,YYY` 字段，那么 `CC` 字段会被忽略。 如果 `CC` 与 `YY` 或 `Y` 一起使用， 那么年份用指定的世纪计算。如果指定了世纪而没有指定年，那么假设使用这个世纪的第一年。
- ISO周时间（有别于公历日期）可以用下面的两种方法之一声明为 `to_timestamp` 和 `to_date` ：
    - 年，周和工作日：例如 `to_date('2006-42-4', 'IYYY-IW-ID')` 返回日期 `2006-10-19` 。如果你省略工作日，那么它假设为1（星期一）。
    - 年和一年中的日：例如 `to_date('2006-291', 'IYYY-IDDD')` 返回 `2006-10-19` 。

  试图用ISO周和公历日期字段混合构造日期是没有意义的，并且将导致一个错误。 在ISO年的范围，"月"和"月中的天"的概念没有意义。 在公历年的范围，ISO周没有意义。用户应该避免混合公历和ISO日期规范。

- 将字符串转化为 `timestamp` 时，毫秒(`MS`)和微秒(`US`) 都是用小数点后面的位数转换的。比如 `to_timestamp('12:3', 'SS:MS')` 不是 3 毫秒, 而是 300 毫秒，因为转换把它看做 12+0.3 秒。这意味着对于格式 `SS:MS` 而言，输入值 `12:3` , `12:30` , `12:300` 声明了相同的毫秒数。对于 3 毫秒，你必须使用 `12:003` ， 那么转换会把它看做 12+0.003 = 12.003 秒。

  这个更复杂的例子 `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` 是 15小时、12分钟、2秒+20毫秒+1230微秒 = 2.021230秒。

- `to_char(..., 'ID')` 的星期编号匹配 `extract(isodow from ...)` 函数，但是 `to_char(..., 'D')` 的星期编号不匹配 `extract(dow from ...)` 的天编号。
- `to_char(interval)` 将 `HH` 和 `HH12` 格式化为12小时， 也就是零时和36时输出是 `12` ，而 `HH24` 可以输出完整的小时数， 时间间隔可以超过23。

Table 9-23显示了用于数值格式化的模板模式。

Table 9-23. 数值格式化的模版模式

模式	描述
`9`	带有指定数值位数的值
`0`	带前导零的值
`.` (句点)	小数点
`,` (逗号)	分组(千) 分隔符
`PR`	尖括号内负值
`S`	带符号的数值(使用区域设置)
`L`	货币符号(使用区域设置)
`D`	小数点(使用区域设置)
`G`	分组分隔符(使用区域设置)
`MI`	在指明的位置的负号(如果数字 < 0)
`PL`	在指明的位置的正号(如果数字 > 0)
`SG`	在指明的位置的正/负号
`RN`	罗马数字(输入在 1 和 3999 之间)
`TH` or `th`	序数后缀
`V`	移动指定位(小数)(参阅注解)
`EEEE`	指数为科学记数法

数字格式化的用法须知：

- 使用 `SG`，`PL`，`MI` 生成的符号并不挂在数字上面；比如，`to_char(-12, 'MI9999')` 生成 `'- 12'`；但是 `to_char(-12, 'S9999')` `' -12'`。Oracle里的实现不允许在 `9` 前面使用 `MI`，而是要求 `9` 在 `MI` 前面。
- `9` 声明和 `9` 的个数相同的数字位数的数值。如果某个数值位没有数字，则输出一个空白。
- `TH` 不会转换小于零的数值，也不会转换小数。
- `PL`，`SG`，`TH` 是PostgreSQL扩展。
- `V` 方便地把输入值乘以 `10^``_n_`，这里 `_n_` 是跟在 `V` 后面的数字。`to_char` 不支持把 `V` 与一个小数点组合在一起使用(也就是说 `99.9V99` 是不允许的)。
- `EEEE`（科学记数法）不能和任何其他格式化的模式或修饰符以外的数字和小数点模式混合使用，并且必须在格式化字符串的后面（例如，`9.99EEEE` 是合法的模式）。

一定的修饰符可以应用于任何模板模式来改变其行为。例如， `FM9999` 是 `9999` 模式和 `FM` 修饰符。 Table 9-24显示了数字格式的修饰符模式。

Table 9-24. 数字格式的模板模式修饰符

修饰符	描述	示例
`FM` 前缀	填充模式 (抑制填充空白和尾随零)	`FM9999`
`TH` 后缀	大写顺序数后缀	`999TH`
`th` 后缀	小写顺序数后缀	`999th`

Table 9-25显示了一些 `to_char` 函数的用法。

Table 9-25. `to_char` 示例

表达式	结果
`to_char(current_timestamp, 'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')`	`'Tuesday, 6 05:39:18'`
`to_char(-0.1, '99.99')`	`' -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(0.1, '0.9')`	`' 0.1'`
`to_char(12, '9990999.9')`	`' 0012.0'`
`to_char(12, 'FM9990999.9')`	`'0012.'`
`to_char(485, '999')`	`' 485'`
`to_char(-485, '999')`	`'-485'`
`to_char(485, '9 9 9')`	`' 4 8 5'`
`to_char(1485, '9,999')`	`' 1,485'`
`to_char(1485, '9G999')`	`' 1 485'`
`to_char(148.5, '999.999')`	`' 148.500'`
`to_char(148.5, 'FM999.999')`	`'148.5'`
`to_char(148.5, 'FM999.990')`	`'148.500'`
`to_char(148.5, '999D999')`	`' 148.500'`
`to_char(3148.5, '9G999D999')`	`' 3,148.500'`
`to_char(-485, '999S')`	`'485-'`
`to_char(-485, '999MI')`	`'485-'`
`to_char(485, '999MI')`	`'485 '`
`to_char(485, 'FM999MI')`	`'485'`
`to_char(485, 'PL999')`	`'+485'`
`to_char(485, 'SG999')`	`'+ 485'`
`to_char(-485, 'SG999')`	`'-485'`

to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. 时间/日期函数和操作符

Table 9-27显示了可以用于处理日期/时间数值的函数， 随后一节里描述了细节。Table 9-26演示了基本算术操作符的行为(`+` , `*` , 等)。 而与格式化相关的函数，可以参考Section 9.8。你应该很熟悉Section 8.5的日期/时间数据类型的背景知识。

所有下述函数和操作符接收的 `time` 或 `timestamp` 输入实际上都来自两种可能： 一种是接收 `time with time zone` 或 `timestamp with time zone` ， 另外一种是接收 `time without time zone` 或 `timestamp without time zone` 。 出于简化考虑，这些变种没有独立显示出来。还有， `+` 和 `*` 操作符都是以可交换的操作符对(比如，date + integer 和 integer + date)；我们只显示了这样的交换操作符对中的一个。

Table 9-26. 日期/时间操做符

操作符	例子	结与
`+`	`date '2001-09-28' + integer '7'`	`date '2001-10-05'`
`+`	`date '2001-09-28' + interval '1 hour'`	`timestamp '2001-0`
`+`	`date '2001-09-28' + time '03:00'`	`timestamp '2001-0`
`+`	`interval '1 day' + interval '1 hour'`	`interval '1 day 0`
`+`	`timestamp '2001-09-28 01:00' + interval '23 hours'`	`timestamp '2001-0`
`+`	`time '01:00' + interval '3 hours'`	`time '04:00:00'`
`-`	`- interval '23 hours'`	`interval '-23:00:`
`-`	`date '2001-10-01' - date '2001-09-28'`	`integer '3'` (day
`-`	`date '2001-10-01' - integer '7'`	`date '2001-09-24'`
`-`	`date '2001-09-28' - interval '1 hour'`	`timestamp '2001-0`
`-`	`time '05:00' - time '03:00'`	`interval '02:00:0`
`-`	`time '05:00' - interval '2 hours'`	`time '03:00:00'`
`-`	`timestamp '2001-09-28 23:00' - interval '23 hours'`	`timestamp '2001-0`
`-`	`interval '1 day' - interval '1 hour'`	`interval '1 day -`
`-`	`timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'`	`interval '1 day 1`
`*`	`900 * interval '1 second'`	`interval '00:15:0`
`*`	`21 * interval '1 day'`	`interval '21 days`
`*`	`double precision '3.5' * interval '1 hour'`	`interval '03:30:0`
`/`	`interval '1 hour' / double precision '1.5'`	`interval '00:40:0`

Table 9-27. 日期/时间函数

函数	返回类型	描述
`age(``timestamp, timestamp)	interval	减去参数后的"符号化"结
`age(``timestamp)	interval	从 current_date 减去参（在午夜）
clock_timestamp()	timestamp with time zone	实时时钟的当前时间戳时变化）；见Section 9.
current_date	date	当前的日期；见Section
current_time	time with time zone	当日时间；见Section 9.
current_timestamp	timestamp with time zone	当前事务开始时的时间戳 Section 9.9.4
`date_part(``text, timestamp)	double precision	获取子域(等效于 extrac Section 9.9.1
`date_part(``text, interval)	double precision	获取子域(等效于 extrac Section 9.9.1
`date_trunc(``text, timestamp)	timestamp	截断成指定的精度；又 9.9.2
extract`( field from timestamp`)	double precision	获取子域；又见Section
extract`( field from interval`)	double precision	获取子域；又见Section
`isfinite(``date)	boolean	测试是否为有穷日期(不
`isfinite(``timestamp)	boolean	测试是否为有穷时间戳(穷)
`isfinite(``interval)	boolean	测试是否为有穷时间间
`justify_days(``interval)	interval	按照每月 30 天调整时间
`justify_hours(``interval)	interval	按照每天 24 小时调整时
`justify_interval(``interval)	interval	使用 justify_days 和 just 整时间间隔的同时进行
localtime	time	当日时间；见Section 9.
localtimestamp	timestamp	当前事务开始时的时间戳 Section 9.9.4
now()	timestamp with time zone	当前事务开始时的时间戳 Section 9.9.4
statement_timestamp()	timestamp with time zone	实时时钟的当前时间戳 9.9.4
timeofday()	text	与 clock_timestamp 相同一个 text 字符串；见S
transaction_timestamp()	timestamp with time zone	当前事务开始时的时间戳

`transaction_timestamp()`	`timestamp with time zone`	Section 9.9.4

除了这些函数以外，还支持 SQL 的 `OVERLAPS` 操作符：

```
(_start1_, _end1_) OVERLAPS (_start2_, _end2_)
(_start1_, _length1_) OVERLAPS (_start2_, _length2_)
```

这个表达式在两个时间域(用它们的终点定义)重叠的时候生成真值，在不重叠是生成假值。终点可以用一对日期、时间、时间戳来声明；或者是一个后面跟着一个时间间隔的日期、时间、时间戳。当提供一对值，不管先写开始还是结束；`OVERLAPS` 自动将这对值较早的作为开始。每段时间取值为半开区间 `_开始_` `<=` `_时间_` `<` `_结束_`，除非 `_开始_` 和 `_结束_` 相等，此时表示单一的时刻。这意味着两个时间段只有一个共同的端点没有重叠。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
_Result:_ <samp class="literal">true</samp>
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
_Result:_ <samp class="literal">false</samp>
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
_Result:_ <samp class="literal">false</samp>
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
_Result:_ <samp class="literal">true</samp>
```

当把 `interval` 值添加到 `timestamp with time zone` 上(或从中减去)的时候，days 部分会按照指定的天数增加(或减少) `timestamp with time zone` 的日期。对于横跨夏令时的变化(会话的时区设置被识别为夏时制)，`interval '1 day'` 并不一定等于 `interval '24 hours'`。例如，当会话的时区设置为 `CST7CDT` 的时候 `timestamp with time zone '2005-04-02 12:00-07' + interval '1 day'` 的结果是 `timestamp with time zone '2005-04-03 12:00-06'`，而将 `interval '24 hours'` 增加到相同的 `timestamp with time zone` 之上的结果则是 `timestamp with time zone '2005-04-03 13:00-06'`，因为 `CST7CDT` 时区在 `2005-04-03 02:00` 的时候有一个夏令时变更。

注意 `age` 返回的 `月数` 可能有歧义，因为不同的月份有不同的天数。PostgreSQL的方法是当计算部分月数时，采用两个日期较早的月。例如：`age('2004-06-01', '2004-04-30')` 使用4月份产生 `1 mon 1 day`，当用5月分时产生 `1 mon 2 days`，因为5月有31天，而4月只有30天。

9.9.1. `EXTRACT`, `date_part`

```
EXTRACT(_field_ FROM _source_)
```

`extract` 函数从日期/时间数值里抽取子域，比如年、小时等。 `_source_` 必须是一个 `timestamp` ， `time` ， `interval` 类型的值表达式(类型为 `date` 的表达式转换为 `timestamp` ，因此也可以用)。 `_field_` 是一个标识符或者字符串，它指定从源数据中抽取的域。 `extract` 函数返回类型为 `double precision` 的数值。下列数值是有效数据域的名字：

`century`

世纪

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
_Result:_ &lt;samp class="literal"&gt;20&lt;/samp&gt;
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;21&lt;/samp&gt;
```

第一个世纪从 0001-01-01 00:00:00 AD 开始，尽管那时候人们还不知道这是第一个世纪。 这个定义适用于所有使用阳历的国家。没有 0 世纪，我们直接从公元前 1 世纪到公元 1 世纪。如果你认为这个不合理，那么请把抱怨发给：梵蒂冈，罗马圣彼得教堂，教皇收。

PostgreSQL 8.0 以前版本里并不遵循世纪的习惯编号， 只是把年份除以 100 。

`day`

对于 `timestamp` 值，(月份)里的日期(1-31)；对于 `interval` ， 天数

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;16&lt;/samp&gt;

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
_Result:_ &lt;samp class="literal"&gt;40&lt;/samp&gt;
```

`decade`

年份除以 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;200&lt;/samp&gt;
```

`dow`

每周的星期号，星期天(`0`)到星期六(`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;5&lt;/samp&gt;
```

请注意， `extract` 的星期几编号和 `to_char(..., 'D')` 函数不同。

`doy`

一年的第几天(1-365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;47&lt;/samp&gt;
```

`epoch`

对于 `timestamp with time zone` 值而言， 是自 1970-01-01 00:00:00-00 UTC以来的秒数(结果可能是负数)； 对于 `date` 和 `timestamp` 值而言， 是自 1970-01-01 00:00:00 当地时间以来的秒数； 对于 `interval` 值而言，它是时间间隔的总秒数。

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
_Result:_ &lt;samp class="literal"&gt;982384720.12&lt;/samp&gt;

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
_Result:_ &lt;samp class="literal"&gt;442800&lt;/samp&gt;
```

下面是把 epoch 值转换回时间戳的方法：

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second';
```

（ `to_timestamp` 函数封装上面的转换。）

`hour`

小时域(0-23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;20&lt;/samp&gt;
```

`isodow`

周中的第几天 [1-7] 星期一：(`1`)星期天：(`7`)。

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
_Result:_ &lt;samp class="literal"&gt;7&lt;/samp&gt;
```

除了星期天外，都与 `dow` 相同。这与ISO 8601标准周中的第几天编码相匹配。

`isoyear`

日期中的ISO 8601标准年（不适用于间隔）。

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
_Result:_ &lt;samp class="literal"&gt;2005&lt;/samp&gt;
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
_Result:_ &lt;samp class="literal"&gt;2006&lt;/samp&gt;
```

每个带有星期一开始的周中包含1月4日的ISO年， 所以在年初的1月或12月下旬的ISO年可能会不同于阳历的年。 见 `week` 获取更多信息。

这个域不能用于 PostgreSQL 8.3之前的版本。

`microseconds`

秒域(包括小数部分)乘以 1,000,000 。请注意它包括全部的秒。

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
_Result:_ &lt;samp class="literal"&gt;28500000&lt;/samp&gt;
```

`millennium`

千年

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;3&lt;/samp&gt;
```

20 世纪(19xx 年)里面的年份在第二个千年里。第三个千年从 2001 年 1 月 1 日零时开始。

PostgreSQL 8.0 之前的版本并不遵循千年编号的习惯， 只是返回年份除以 1000 。

`milliseconds`

秒域(包括小数部分)乘以 1000 。请注意它包括完整的秒。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
_Result:_ &lt;samp class="literal"&gt;28500&lt;/samp&gt;
```

`minute`

分钟域(0-59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;38&lt;/samp&gt;
```

`month`

对于 `timestamp` 值，它是一年里的月份数(1-12)；对于 `interval` 值， 它是月的数目，然后对 12 取模(0-11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;2&lt;/samp&gt;

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
_Result:_ &lt;samp class="literal"&gt;3&lt;/samp&gt;

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
_Result:_ &lt;samp class="literal"&gt;1&lt;/samp&gt;
```

`quarter`

该天所在的该年的季度(1-4)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;1&lt;/samp&gt;
```

`second`

秒域，包括小数部分(0-59) [1]

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;40&lt;/samp&gt;

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
_Result:_ &lt;samp class="literal"&gt;28.5&lt;/samp&gt;
```

`timezone`

与 UTC 的时区偏移量，以秒记。正数对应 UTC 东边的时区，负数对应 UTC 西边的时区。（技术角度讲，PostgreSQL使用UT1，因为不处理闰秒。）

`timezone_hour`

时区偏移量的小时部分。

`timezone_minute`

时区偏移量的分钟部分。

`week`

该天在所在的年份里是第几周。ISO 8601 定义一年的第一周包含该年的一月四日(ISO-8601 的周从星期一开始)。换句话说， 一年的第一个星期四在第一周。

在ISO定义里，一月的头几天可能是前一年的第 52 或者第 53 周， 十二月的后几天可能是下一年第一周。比如， `2005-01-01` 是 2004 年的第 53 周， 而 `2006-01-01` 是 2005 年的第 52 周， `2012-12-31` 是2013年的第一周。 建议 `isoyear` 字段和 `week` 一起使用以得到一致的结果。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;7&lt;/samp&gt;
```

`year`

年份域。要记住这里没有 `0 AD` ，所以从 `AD` 年里抽取 `BC` 年应该小心些。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
_Result:_ &lt;samp class="literal"&gt;2001&lt;/samp&gt;
```

`extract` 函数主要的用途是运算。 对于用于显示的日期/时间数值格式化，参阅Section 9.8。

`date_part` 函数是在传统的Ingres 函数的基础上制作的(该函数等效于SQL标准函数 `extract`)：

```
date_part('_field_', _source_)
```

请注意这里的 `_field_` 参数必须是一个字符串值，而不是一个名字。 有效的 `date_part` 数域名和 `extract` 是一样的。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
_Result:_ <samp class="literal">16</samp>

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
_Result:_ <samp class="literal">4</samp>
```

9.9.2. `date_trunc`

`date_trunc` 函数在概念上和用于数字的 `trunc` 函数类似。

```
date_trunc('_field_', _source_)
```

`_source_` 是 `timestamp` 或 `interval` 类型的值表达式(`date` 和 `time` 类型的值都分别自动转换成 `timestamp` 或 `interval`)。用 `_field_` 选择对该时间戳值用什么样的精度进行截断。返回的数值是 `timestamp` 或 `interval` 类型，所有小于选定的精度的域都设置为零(日期和月份域则为 1)。

`_field_` 的有效数值是：

`microseconds`
`milliseconds`
`second`
`minute`
`hour`
`day`
`week`
`month`
`quarter`
`year`
`decade`
`century`
`millennium`

例子：

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
_Result:_ <samp class="literal">2001-02-16 20:00:00</samp>

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
_Result:_ <samp class="literal">2001-01-01 00:00:00</samp>
```

9.9.3. `AT TIME ZONE`

`AT TIME ZONE` 构造允许把时间戳转换成不同的时区。 Table 9-28显示了其变体。

Table 9-28. `AT TIME ZONE` 变体

表达式	返回类型	描述
`timestamp without time zone` AT TIME ZONE `_zone_`	`timestamp with time zone`	把给出的不带时区的时间戳转换成给定时区的时间戳
`timestamp with time zone` AT TIME ZONE `_zone_`	`timestamp without time zone`	把给出的带时区的时间戳转换成未指定时区的时间戳
`time with time zone` AT TIME ZONE `_zone_`	`time with time zone`	把给出的带时区的时间转换成给定时区的时间

在这些表达式里， `_zone_` 可以声明为文本串(比如 `'PST'`) 或者一个时间间隔(比如 `INTERVAL '-08:00'`)。在文本的情况下， 可用的时区名字在Section 8.5.3有详细描述。

例子(假设本地时区是 `PST8PDT`)：

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
_Result:_ <samp class="literal">2001-02-16 19:38:40-08</samp>

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
_Result:_ <samp class="literal">2001-02-16 18:38:40</samp>
```

第一个例子接受一个无时区的时间戳然后把它解释成 MST(UTC-7) 时间生成 UTC 时间戳, 然后把这个时间转换为 PST(UTC-8) 显示。第二个例子接受一个声明为 EST(UTC-5) 的时间戳， 然后把它转换成 MST(UTC-7) 的当地时间。

`timezone` (`_zone_` , `_timestamp_`) 函数等效于 SQL 兼容的构造 `_timestamp_` AT TIME ZONE `_zone_` 。

9.9.4. 当前日期/时间

PostgreSQL提供许多返回当前日期和时间的函数。 这些符合 SQL 标准的函数全部都按照当前事务的开始时刻返回结果：

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(_precision_)
CURRENT_TIMESTAMP(_precision_)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(_precision_)
LOCALTIMESTAMP(_precision_)
```

`CURRENT_TIME` 和 `CURRENT_TIMESTAMP` 返回带有时区的值； `LOCALTIME` 和 `LOCALTIMESTAMP` 返回不带时区的值。

`CURRENT_TIME` , `CURRENT_TIMESTAMP` , `LOCALTIME` , `LOCALTIMESTAMP` 可以有选择地获取一个精度参数，该精度导致结果的秒数域园整到指定小数位。 如果没有精度参数，将给予所能得到的全部精度。

一些例子：

```
SELECT CURRENT_TIME;
_Result:_ <samp class="literal">14:39:53.662522-05</samp>

SELECT CURRENT_DATE;
_Result:_ <samp class="literal">2001-12-23</samp>

SELECT CURRENT_TIMESTAMP;
_Result:_ <samp class="literal">2001-12-23 14:39:53.662522-05</samp>

SELECT CURRENT_TIMESTAMP(2);
_Result:_ <samp class="literal">2001-12-23 14:39:53.66-05</samp>

SELECT LOCALTIMESTAMP;
_Result:_ <samp class="literal">2001-12-23 14:39:53.662522</samp>
```

因为这些函数全部都按照当前事务的开始时刻返回结果， 所以它们的值在事务运行的整个期间内都不改变。我们认为这是一个特性： 目的是为了允许一个事务在"当前时间"上有连贯的概念， 这样在同一个事务里的多个修改可以保持同样的时间戳。

> **Note:** 许多其它数据库系统更频繁地更新这些数值。

PostgreSQL同样也提供了返回实时时间值的函数， 它们的返回值会在事务中随时间的前进而变化。这些不附合 SQL 标准的函数列表如下：

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` 等效于 `CURRENT_TIMESTAMP` ， 不过其命名准确的表明了其含义。 `statement_timestamp()` 返回当前事务开始时刻的时间戳(更准确的说是收到客户端最后一条命令的时间)。 `statement_timestamp()` 和 `transaction_timestamp()` 在一个事务的第一条命令里返回值相同，但是在随后的命令中却不一定相同。 C `clock_timestamp()` 返回实时时钟

的当前时间戳， 因此它的值甚至在同一条 SQL 命令中都会变化。 `timeofday()` 是一个历史的PostgreSQL函数，类似于 `clock_timestamp()` ， 它也返回实时时钟的当前时间戳，不过它返回一个格式化了的 `text` 字符串， 而不是 `timestamp with time zone` 值。 `now()` 是传统的PostgreSQL和 `transaction_timestamp()` 等效的函数。

所有日期/时间类型还接受特殊的文本值 `now`， 用于声明当前的日期和时间(重申：当前事务的开始时刻)。 因此，下面三个都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- incorrect for use with DEFAULT
```

> **Tip:** 在创建表的时候你不应该用第三种形式声明一个 `DEFAULT` 值。 系统将在分析这个常量的时候把 `now` 转换为一个 `timestamp`， 因此这个缺省值就会变成创建表的时间！而前两种形式要到实际使用缺省值的时候才计算， 因为它们是函数调用。因此它们可以给出每次插入行的时刻。

9.9.5. 延时执行

下面的这个函数可以用于让服务器进程延时执行：

```
pg_sleep(_seconds_)
```

`pg_sleep` 让当前的会话进程休眠 `_seconds_` 秒以后再执行。 `_seconds_` 是一个 `double precision` 类型的值，所以可以指定带小数的秒数。例如：

```
SELECT pg_sleep(1.5);
```

> **Note:** 有效的休眠时间间隔精度是平台相关的，通常 0.01 秒是通用的。 休眠的时间将至少等于指定的时间，也有可能由于服务器荷载较重等原因而比指定的时间长。

Warning
请确保调用 `pg_sleep` 的会话没有持有不必要的锁。 否则其它会话可能必须等待这个休眠的会话释放所持有的锁，从而减慢系统速度。

Notes

[1] 如果操作系统实现了润秒，那么上限是 60 。

9.10. 支持枚举函数

对于枚举类型(描述在Section 8.7)， 有一些函数允许简洁的编写没有硬编码特定的枚举类型值。 这些列在Table 9-29。像下面的创建枚举类型的例子：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

Table 9-29. 支持枚举函数

函数	描述	
`enum_first(anyenum)`	返回输入枚举类型的第一个值	`enum_firs`
`enum_last(anyenum)`	返回输入枚举类型的最后一个值	`enum_last`
`enum_range(anyenum)`	以一个有序的数组的形式返回输入枚举类型的所有值	`enum_rang`
`enum_range(anyenum, anyenum)`	返回在给定两个枚举值之间的范围，作为一个有序数组。该值必须是相同的枚举类型。 如果第一个参数为空，其结果将从枚举类型的第一个值开始。 如果第二参数为空，其结果将以枚举类型的最后一个值结束。	`enum_rang`
`enum_range(NULL, 'green'::rainbow)`	`{red,orange,yellow,green}`	
`enum_range('orange'::rainbow, NULL)`	`{orange,yellow,green,blue,purple}`	

请注意，除了两个参数形式的 `enum_range` 外，这些函数忽略传递给他们的具体值， 他们只关心声明的数据类型。null或一个特定类型的值可以通过，得到相同的结果。 更常见于这些功能应用于表列或函数的参数而不是一个硬性的类型名称， 就像例子中建议的那样。

9.11. 几何函数和操作符

有许多内置函数和操作符支持几何类型 `point` , `box` , `lseg` , `line` , `path` , `polygon` , `circle` ， 在Table 9-30, Table 9-31, Table 9-32中展示。

Caution

请注意"相同"操作符 `~=` 表示 `point` , `box` , `polygon` , `circle` 类型在一般意义上相同。 这些类型有些还有一个 `=` 操作符，不过它只是比较相同的面积。 其它的标量比较操作符(`<=` 等)也是为这些类型比较面积。

Table 9-30. 几何操作符

操作符	描述	例子
`+`	平移	`box '((0,0),(1,1))' + point '(2.0,0)'`
`-`	平移	`box '((0,0),(1,1))' - point '(2.0,0)'`
`*`	伸缩/旋转	`box '((0,0),(1,1))' * point '(2.0,0)'`
`/`	伸缩/旋转	`box '((0,0),(2,2))' / point '(2.0,0)'`
`#`	交点或者交面	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
`#`	路径或多边形顶点数	`# '((1,0),(0,1),(-1,0))'`
`@-@`	长度或者周长	`@-@ path '((0,0),(1,0))'`
`@@`	中心	`@@ circle '((0,0),10)'`
`##`	第一个操作数相对第二个操作数的最近点	`point '(0,0)' ## lseg '((2,0),(0,2))'`
`<->`	间距	`circle '((0,0),1)' <-> circle '((5,0),1)'`
`&&`	重叠？（有一个共同点为真。）	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
	是否严	

<<	格在左?	circle '((0,0),1)' << circle '((5,0),1)'
>>	是否严格在右?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	是否没有延伸到右边?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	是否没有延伸到左边?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	严格在下?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	严格在上?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	没有延伸到上面?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	没有延伸到下面?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	低于(允许接触)?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	高于(允许接触)?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	相交?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	水平?	?- lseg '((-1,0),(1,0))'
?-	水平对齐?	point '(1,0)' ?- point '(0,0)'
?|	竖直?	?| lseg '((-1,0),(1,0))'
?|	竖直对齐?	point '(0,1)' ?| point '(0,0)'
?-|	垂直?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	平行?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	包含?	circle '((0,0),2)' @> point '(1,1)'
<@	包含或在...	point '(1,1)' <@ circle '((0,0),2)'

	上?	
`~=`	与...相同?	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

> **Note:** 在PostgreSQL 8.2 之前，包含操作符 `@>` 和 `<@` 被分别称为 `~` 和 `@` 。 我们反对使用这两个旧名字(当前仍然可以使用)，它们将来会被废除。

Table 9-31. 几何函数

函数	返回类型	描述	例子
`area(```_object_)	`double precision`	面积	`area(box '((0,0),(1,1))')`
`center(```_object_)	`point`	中心	`center(box '((0,0),(1,2))')`
`diameter(```circle)	`double precision`	圆直径	`diameter(circle '((0,0),2.0)')`
`height(```box)	`double precision`	矩形的竖直高度	`height(box '((0,0),(1,1))')`
`isclosed(```path)	`boolean`	闭合路径?	`isclosed(path '((0,0),(1,1),(2,0))`
`isopen(```path)	`boolean`	开路径?	`isopen(path '[(0,0),(1,1),(2,0)]')`
`length(```_object_)	`double precision`	长度	`length(path '((-1,0),(1,0))')`
`npoints(```path)	`int`	点数	`npoints(path '[(0,0),(1,1),(2,0)]'`
`npoints(```polygon)	`int`	点数	`npoints(polygon '((1,1),(0,0))')`
`pclose(```path)	`path`	把路径转换为闭	`pclose(path '[(0,0),(1,1),(2,0)]')`

		合	
`popen(```path)	path	把路径转换为开放	popen(path '((0,0),(1,1),(2,0))')
`radius(```circle)	double precision	圆半径	radius(circle '((0,0),2.0)')
`width(```box)	double precision	矩形的水平尺寸	width(box '((0,0),(1,1))')

Table 9-32. 几何类型转换函数

函数	返回类型	描述	例子
`box(```circle)	box	将圆转换成矩形	box(circle '((0,0),2.0)')
`box(```point , point)	box	将点转换成矩形	box(point '(0,0)', point '(1,1)')
`box(```polygon)	box	将多边形转换成矩形	box(polygon '((0,0),(1,1),(2,0))')
`circle(```box)	circle	矩形转换成圆	circle(box '((0,0),(1,1))')
`circle(```point , double precision)	circle	将圆心和半径转换成圆	circle(point '(0,0)', 2.0)
`circle(```polygon)	circle	将多边形转换成圆	circle(polygon '((0,0),(1,1),(2,0)
`lseg(```box)	lseg	矩形对角线转化成线段	lseg(box '((-1,0),(1,0))')
`lseg(```point , point)	lseg	点转换成线段	lseg(point '(-1,0)', point '(1,0)'
`path(```polygon)	path	多边形转换成路径	path(polygon '((0,0),(1,1),(2,0))'
point`( double precision , double precision`)	point	结点	point(23.4, -44.5)
`point(```box)	point	矩形的中心	point(box '((-1,0),(1,0))')
`point(```circle)	point	圆心	point(circle '((0,0),2.0)')
`point(```lseg)	point	线段的中心	point(lseg '((-1,0),(1,0))')
`point(```polygon)	point	多边形的中心	point(polygon '((0,0),(1,1),(2,0))'
`polygon(```box)	polygon	矩形转换成4 点多边形	polygon(box '((0,0),(1,1))')
`polygon(```circle)	polygon	圆转换成 12 点多边形	polygon(circle '((0,0),2.0)')
`polygon(```_npts_ , circle)	polygon	圆转换成 _npts_ 点多边形	polygon(12, circle '((0,0),2.0)')
`polygon(```path)	polygon	路径转换成多边形	polygon(path '((0,0),(1,1),(2,0))'

我们可以把一个`point`的两个组成部分当作索引分别为 0 和 1 的数组元素进行访问。 比如，如果`t.p`是一个`point`字段，那么`SELECT p[0] FROM t` 检索 X 座标而`UPDATE t SET p[1] = ...`改变 Y 座标。同样， `box`或`lseg`的值可以当作两个`point`的数组值看待。

`area`函数可以用于`box`，`circle`，`path`类型。`area`函数操作`path`数据类型的时候， 只有在`path`的点没有交叉的情况下才可用。比如， `path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH`是不行的， 而下面的视觉等效`path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH` 就可以。如果交叉和不交叉的`path`概念让你糊涂，那么把上面两个`path` 都画在纸上，你就明白了。

9.12. 网络地址函数和操作符

Table 9-33显示了可以用于 cidr 和 inet 的操作符。操作符 << ， <<= ， >> ， >>= 用于测试子网包含： 它们只考虑两个地址的网络部分， 忽略任何主机部分， 然后判断其中一个网络是等于另外一个还是另外一个的子网。

Table 9-33. cidr 和 inet 操作符

操作符	描述	例子
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于或等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于或等于	inet '192.168.1.5' >= inet '192.168.1.5'
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	包含于	inet '192.168.1.5' << inet '192.168.1/24'
<<=	包含于或等于	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	包含	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'
~	位非	~ inet '192.168.1.6'
&	位与	inet '192.168.1.6' & inet '0.0.0.255'
		位或	inet '192.168.1.6'	inet '0.0.0.255'
+	加	inet '192.168.1.6' + 25
-	减	inet '192.168.1.43' - 36
-	减	inet '192.168.1.43' - inet '192.168.1.19'

Table 9-34显示了所有可以用于 cidr 和 inet 的函数。函数 abbrev ， host ， text 主要是为了提供可选的显示格式用的。

Table 9-34. cidr 和 inet 函数

函数	返回类型	描述	例子
`abbrev(``inet)`	text	缩写显示格式文本	abbrev(inet '10.1.0.0/16')
`abbrev(``cidr)`	text	缩写显示格式文本	abbrev(cidr '10.1.0.0/16')
`broadcast(``inet)`	inet	网络广播地址	broadcast('192.168.1.5/24')
`family(``inet)`	int	抽取地址族; 4 为 IPv4, 6 为 IPv6	family('::1')
`host(``inet)`	text	将主机地址类型抽出为文本	host('192.168.1.5/24')
`hostmask(``inet)`	inet	为网络构造主机掩码	hostmask('192.168.23.20/30')
`masklen(``inet)`	int	抽取子网掩码长度	masklen('192.168.1.5/24')
`netmask(``inet)`	inet	为网络构造子网掩码	netmask('192.168.1.5/24')
`network(``inet)`	cidr	抽取地址的网络部分	network('192.168.1.5/24')
`set_masklen(``inet , int)`	inet	为 inet 数值设置子网掩码长度	set_masklen('192.168.1.5/24', 16)
`set_masklen(``cidr , int)`	cidr	为 cidr 数值设置子网掩码长度	set_masklen('192.168.1.0/24'::cidr, 16
`text(``inet)`	text	把 IP 地址和掩码长度抽取为文本	text(inet '192.168.1.5')

任何 `cidr` 值都能够被隐含或明确的转换为 `inet` 值， 因此上述能够操作 `inet` 值的函数也同样能够操作 `cidr` 值。 而将某些操作 `inet` 和 `cidr` 的函数单独分隔开是因为它们的行为不同。 `inet` 值也可以转换为 `cidr` 值， 此时子网掩码右侧的所有位都将无声的转换为零以获

得一个有效的 `cidr` 值。 另外，你还可以使用常规的类型转换语法将一个文本字符串转换为 `inet` 或 `cidr` 值。 例如： ``inet(``_expression_``)或 `_colname_` ::cidr。

Table 9-35显示了可以用于 `macaddr` 类型的函数。 函数 `` `trunc(```macaddr ``)返回一个 MAC 地址， 该地址的最后三个字节设置为零。这样可以把剩下的前缀与一个制造商相关联。

Table 9-35. `macaddr` 函数

函数	返回类型	描述	例子	
`` `trunc(```macaddr ``)	`macaddr`	把后三个字节置为零	`trunc(macaddr '12:34:56:78:90:ab')`	`12:34:`

`macaddr` 类型还支持标准关系操作符(> , <= 等)用于词法排序，和按位运算符(`~` , `&` 和 `|`)非，与和或。

9.13. 文本检索函数和操作符

Table 9-36，Table 9-37 和Table 9-38概述了提供全文检索函数和操作符。 详见Chapter 12 PostgreSQL文本检索功能的说明。

Table 9-36. 文本检索操作符

操作符	描述	示例
`@@`	`tsvector` 匹配 `tsquery` ?	`to_tsvector('fat cats ate rats') @@ to_tsquery('cat &`
`@@@`	弃用的 `@@` 的同义词	`to_tsvector('fat cats ate rats') @@@ to_tsquery('cat &`
`||`	连接 `tsvector` s	`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvect`
`&&`	`tsquery` 与	`'fat	rat'::tsquery && 'cat'::tsquery`
`||`	`tsquery` 或	`'fat	rat'::tsquery || 'cat'::tsquery`
`!!`	`tsquery` 非	`!! 'cat'::tsquery`
`@>`	`tsquery` 包含另一个?	`'cat'::tsquery @> 'cat & rat'::tsquery`
`<@`	`tsquery` 包含于 ?	`'cat'::tsquery <@ 'cat & rat'::tsquery`

> **Note:** `tsquery` 的操作符只考虑两个查询列出的项的情况，忽略组合的操作符。

除了显示在表中的操作符，还为类型 `tsvector` 和 `tsquery` 定义了普通的B-tree比较操作符（ `=` , `<` 等）。 它们对于文本检索不是很有用，但是允许使用。例如，建在这些类型列上的唯一索引。

Table 9-37. 文本检索函数

函数	返回类型	描述	
`get_current_ts_config()`	`regconfig`	获取文本检索的默认配置	`get_current_ts_co`
`` `length(```tsvector) ``	integer	`tsvector` 的单词数	`length('fat:2,4 c`
`` `numnode(```tsquery) ``	`integer`	`tsquery` 中的单词加上操作符的数量	`numnode('(fat & r`
		产	

regconfig ,] _query_ text)	tsquery	生 tsquery 忽略标点	plainto_tsquery('(
`querytree(```_query_ tsquery)	text	获取 tsquery 可索引的部分	querytree('foo &
`setweight(```tsvector , "char")	tsvector	给 tsvector 的每个元素赋予权值	setweight('fat:2,4
`strip(```tsvector)	tsvector	删除 tsvector 中的位置和权值	strip('fat:2,4 cat
`to_tsquery([```_config_ regconfig ,] _query_ text)	tsquery	标准化单词并转换为 tsquery	to_tsquery('englis
`to_tsvector([```_config_ regconfig ,] _document_ text)	tsvector	减少文档中的文本到 tsvector	to_tsvector('engli
`ts_headline([```_config_ regconfig ,] _document_ text , _query_ tsquery [, _options_ text])	text	显示一个查询的匹配项	ts_headline('x y z
`ts_rank([```_weights_ float4[] ,] _vector_ tsvector , _query_ tsquery [, _normalization_ integer])	float4	文档查询排名	ts_rank(textsearch
`ts_rank_cd([```_weights_ float4[] ,] _vector_ tsvector , _query_ tsquery [, _normalization_ integer])	float4	排序文件查询使用覆盖密度	ts_rank_cd('{0.1,
`ts_rewrite(```_query_ tsquery , _target_ tsquery , _substitute_ tsquery)	tsquery	替换带有查询的替代目标	ts_rewrite('a & b
`ts_rewrite(```_query_ tsquery , _select_ text)	tsquery	从一条 SELECT 命令的替代目标做替换	SELECT ts_rewrite(
tsvector_update_trigger()	trigger	自动更新 tsvector 列的触发器函数	CREATE TRIGGER ..
tsvector_update_trigger_column()	trigger	自动更新 tsvector 列的触发器函数	CREATE TRIGGER ..

> **Note:** 所有的文本检索函数，接受一个选项 `regconfig` 参数。当这个参数忽略， 使用由default_text_search_config指定的配置。

Table 9-38单独列出的函数， 因为他们通常不用于每天的文本检索操作。它们有助于开发调试新文本检索配置。

Table 9-38. 文本检索调试函数

函数	返回类型	描述	
`ts_debug([```_config_ regconfig ,] _document_ text , OUT _alias_ text , OUT _description_ text , OUT _token_ text , OUT _dictionaries_ regdictionary[] , OUT _dictionary_ regdictionary , OUT _lexemes_ text[])	setof record	测试一个配置	ts_debug('englis
`ts_lexize(```_dict_ regdictionary , _token_ text)	text[]	测试一个数据字典	ts_lexize('engli
`ts_parse(```_parser_name_ text , _document_ text , OUT _tokid_ integer , OUT _token_ text)	setof record	测试一个解析	ts_parse('defaul
`ts_parse(```_parser_oid_ oid , _document_ text , OUT _tokid_ integer , OUT _token_ text)	setof record	测试一个解析	ts_parse(3722,
`ts_token_type(```_parser_name_ text , OUT _tokid_ integer , OUT _alias_ text , OUT _description_ text)	setof record	由解析器获取分词类型的定义	ts_token_type('d
`ts_token_type(```_parser_oid_ oid , OUT _tokid_ integer , OUT _alias_ text , OUT _description_ text)	setof record	由解析器获取分词类型的定义	ts_token_type(37
`ts_stat(```_sqlquery_ text , [_weights_ text ,] OUT _word_ text , OUT _ndoc_ integer , OUT _nentry_ integer)	setof record	获取 tsvector 列的统计数据	ts_stat('SELECT

9.14. XML 函数

在本节描述的函数和像函数的表达式操作都是基于 `xml` 类型的值。 查看Section 8.13获取关于 `xml` 类型的信息。 像函数表达式的 `xmlparse` 和 `xmlserialize` 用来转换为和从类型 `xml` 转换，不在这里重复。 使用这些函数需要安装与配置了 `configure --with-libxml` 。

9.14.1. 生成XML内容

一组函数和像函数的表达式可用于从SQL数据生成XML内容。 所以它们特别适合于查询结果格式化成在客户端应用程序处理的XML文件。

9.14.1.1. `xmlcomment`

```
xmlcomment(_text_)
```

`xmlcomment` 函数创建一个包含XML注释的特定文本内容的值。 文本中不能包含" `--` "或以" `-` " 的结束，这样的文本是有效的XML注释。如果参数是空，结果是空。

例子：

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.14.1.2. `xmlconcat`

```
xmlconcat(_xml_[, ...])
```

函数 `xmlconcat` 连接一个独立的XML值列表来创建一个包含XML内容片段的单值。 忽略空值；只有当参数都为空时结果是空。

例子：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML声明，如果存在，结合如下。如果所有参数使用相同的XML版本声明，则在结果中使用版本。 否则不用版本。如果所有的参数值有独立的声明值"yes"， 然后这个值在结果里使用。如果所有的参数值有独立的声明，并且至少有一个是"no"， 然后这个值在结果里使用。否则结果将没有独立声明。如果结果决定需要一个独立的声明， 但没有声明版本，将使用一个带有版本1.0的版本声明，因为XML需要一个XML声明包含版本声明。 忽略并且在所有情况下删除编码声明。

例子：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><ba

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. `xmlelement`

```
xmlelement(name _name_ [, xmlattributes(`_value_` [AS `_attname_`] [, ... ])] [`_, conten
```

`xmlelement` 表达式生成一个带有给定名称，属性和内容的XML元素。

例子：

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

不是有效的XML元素和属性名的名称由序列 `_x``_HHHH_` 替换有问题的字符逃逸，这里的 `_*HHHH*` 是字符的16进制形式的Unicode代码点。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用则不用指定明确的属性名称，在这种情况下，列的名称将默认为属性名。 在其它情况下，属性必须给予一个明确的名称。因此，这个例子是有效的：

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但是这些不是:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

如果指定了元素内容，将根据它的数据类型格式化。如果内容自身是 `xml` 类型， 可以构造复杂的xml文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

将其它类型的内容格式化为有效的xml字符串数据。这意味着特殊的字符<, >, 和&将转化为实体。二进制数据（ `bytea` 数据类型） 将用base64或16进制编码表示，取决于配置参数xmlbinary的设置。 单个数据类型的特定行为预计将发展为了使SQL和PostgreSQL数据类型和XML架构规范一致， 到时将出现更准确描述。

9.14.1.4. `xmlforest`

```
xmlforest(_content_ [AS `_name_`] [, ...])
```

`xmlforest` 表达式生成一个使用指定的名称和内容的XML 森林（序列）元素。

示例:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>

SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                         xmlforest
-------------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

在第二个例子可以看出，如果内容值为列引用，元素名称可以省略。在这种情况下， 默认使用列名。否则，必须指定名称。

非法XML名的元素名称，像上面的`xmlelement`逃逸处理。 类似的，内容数据逃逸生成有效的XML内容， 除非它已经是`xml` 类型的。

请注意，如果包含一个以上的元素，XML的森林不是有效的XML文档， 所以在`xmlelement`里面封装`xmlforest`表达式可能是有用的。

9.14.1.5. `xmlpi`

```
xmlpi(name _target_ [, `_content_`])
```

`xmlpi`表达式创建一条XML处理指令。 如果存在，内容必须不能包含字符序列`?>`。

示例：

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.14.1.6. `xmlroot`

```
xmlroot(_xml_, version _text_ | no value [, standalone yes|no|no value])
```

`xmlroot`更改XML值的根节点属性。如果指定一个版本， 它替换根节点的版本声明值;如果指定一个standalone设置，它替换根节点的standalone声明值。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. `xmlagg`

```
xmlagg(_xml_)
```

不像这里描述的其它函数，函数 `xmlagg` 是一个聚集函数。 它连接聚集函数调用的输入值，很像 `xmlconcat` ， 除了连接发生在多行而不是发生在多个单行的表达式。 请参阅Section 9.20获取关于聚集函数的更多信息。

示例：

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

为了确定连接顺序，要添加一个 `ORDER BY` 子句到聚合调用， 描述在Section 4.2.7。示例：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

建议在之前的版本中使用下面非标准的方法，在特例中可能仍然有用:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.2. XML Predicates

这节描述的表达式检查 `xml` 值的属性。

9.14.2.1. `IS DOCUMENT`

```
_xml_ IS DOCUMENT
```

如果参数XML值是一个合法的XML文档，表达式 `IS DOCUMENT` 返回真。 否则返回假（例如，内容片段）或如果参数为空则返回空。请参阅Section 8.13 获取关于文档和内容片段之间的不同。

9.14.2.2. `XMLEXISTS`

```
XMLEXISTS(_text_ PASSING [BY REF] _xml_ [BY REF])
```

如果第一个参数中的XPath表达式返回任何节点，那么函数 `xmlexists` 返回真， 否则返回假。（如果其他参数是null，结果是null。）

示例：

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF '<towns><town>Toronto</tow

 xmlexists
------------
 t
(1 row)
```

在PostgreSQL中， `BY REF` 子句没有影响， 但是为了与SQL的一致性和其他实现的兼容性是允许的。SQL标准中， 第一个 `BY REF` 是必须的，第二个 `BY REF` 是可选的。 也请注意，SQL标准声明 `xmlexists` 构造接受XQuery表达式作为第一个参数， 但是PostgreSQL目前只接受XQuery的一个子集XPath。

9.14.2.3. `xml_is_well_formed`

```
xml_is_well_formed(_text_)
xml_is_well_formed_document(_text_)
xml_is_well_formed_content(_text_)
```

这些函数检查 `text` 字符串是不是格式良好的XML，返回布尔结果。 `xml_is_well_formed_document` 检查格式良好的文档， `xml_is_well_formed_content` 检查格式良好的内容。 `xml_is_well_formed` 如果xmloption参数设置为 `DOCUMENT` 则检查文档，如果设置为 `CONTENT` 则检查内容。 这意味着 `xml_is_well_formed` 有助于看到一个简单到类型 `xml` 的转换是否会成功，而另外两个函数有助于看到相应的 `XMLPARSE` 变体是否会成功。

示例：

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</p
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</m
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

最后一个示例显示了检查包括命名空间是否正确匹配。

9.14.3. 处理XML

PostgreSQL提供了 `xpath` 和 `xpath_exists` 函数处理 `xml` 数据类型的值，计算XPath 1.0表达式的结果。

```
xpath(_xpath_, _xml_ [, `_nsarray_`])
```

`xpath` 函数，对XML值 `_xml_` 计算XPath表达式 `_xpath_` （ `text` 值）的结果。 它返回一个XML值的数组对应XPath表达式所产生的节点集。 如果XPath表达式返回一个标量值而不是节点集，那么返回一个单个元素的数组。

第二个参数必须是一个完整的XML文档。特别是，它必须有一个根节点元素。

该函数的第三个参数是一个命名空间的数组映射。这个数组应该是一个两维 `text` 数组， 第二个维的长度等于2（它应该是一个数组的数组，其中每个正好包含2个元素）。 每个数组项的第一个元素是命名空间名称的别名，第二个元素是命名空间 URI。 这个数组的别名不是必须提供的，与在XML文档本身使用的相同。（换句话说， 在XML文档和在 `xpath` 函数的上下文中， 别名是*local*）。

示例：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

处理默认的命名空间，像下面这样做：

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

```
xpath_exists(_xpath_, _xml_ [, `_nsarray_`])
```

`xpath_exists` 函数是 `xpath` 函数的一种特殊化形式。 这个函数返回一个布尔值表明是否满足这个查询，而不是返回满足XPath的单个XML值。 这个函数相当于标准的 `XMLEXISTS` ，除了它还对命名空间映射参数提供支持。

示例：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                     ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.14.4. 到XML的映射表

下面的函数映射关系表的内容到XML值。可以将它们认为XML导出功能：

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

每个函数的返回类型是 `xml` 。

`table_to_xml` 映射命名表的内容，作为参数 `tbl` 传递。 `regclass` 类型接受使用常用符号的字符串标识表，包括可选的模式资格和双引号。 `query_to_xml` 执行查询，这个查询的文本作为 `query` 参数传递，并映射结果集。 `cursor_to_xml` 从参数 `cursor` 指定的游标中获取指定

数量的行。如果大数据表需要映射，建议使用这个变体， 因为结果值是通过每个函数在内存中构建的。

如果 `tableforest` 是假值，则结果的XML文档像这样：

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

如果 `tableforest` 是真值，结果是一个像这样的XML内容片段：

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

如果没有可用的表名，也就是当映射一个查询或游标时， 第一个格式用字符串 `table` ，第二个格式用 `row` 。

这些格式是给用户选择使用的。第一种格式是适当的XML文档，在许多应用程序中比较重要。 如果结果值是稍后重新组合成一个文件，则第二种格式在 `cursor_to_xml` 函数中更有用。这些函数用来产生上述讨论的XML内容，特别是 `xmlelement` ， 可以用来尝试更改结果。

数据值以上面描述的函数 `xmlelement` 相同的方式映射。

参数 `nulls` 取决于在输出中是否包含空值。如果真，列中的空值表示为：

```
<columnname xsi:nil="true"/>
```

这里的 `xsi` 是XML架构实例的XML命名空间前缀。 将为结果值添加一个适当的命名空间声明。如果假，包含空值的列会从输出中简单的省略。

参数的 `targetns` 指定想要结果的XML命名空间。 如果没有特别想要的命名空间，应传递一个空字符串。

下面的函数返回描述由上述相应的函数执行映射的 XML 架构文档:

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)
```

重要的是要传递相同的参数以获取匹配的XML数据映射和XML架构文档。

下列函数在一个文档（或森林）中生成XML数据映射和相应的XML架构，联系在一起。它们在想要自我包含和自我描述结果的时候可能很有用：

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns tex
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
```

此外，下列函数还可用于生成类似整个模式或整个当前数据库的映射：

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns tex

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

请注意这些可能产生大量的数据，是需要在内存中建立的。当请求大数据量的模式或数据库的内容映射时，可能值得考虑映射表分别替代，可能甚至通过游标。

一个模式内容映射的结果像这样：

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

其中一个表映射的格式取决于上面所述的 `tableforest` 参数。

一个数据库内容映射的结果像这样：

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

模式映射如上所述。

使用这些函数产生的输出作为例子，Figure 9-1显示一个XSLT样式表转换 `table_to_xml_and_xmlschema` 的输出到HTML文档， 该文档中包含了一个表数据的表格格式副本。以类似的方式， 这些函数的结果可以转换成其它基于XML的格式。

Figure 9-1. XSLT样式表--将SQL/XML输出转换成HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequenc
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. JSON 函数和操作符

Table 9-39显示了可以用于JSON （参阅Section 8.14）数据的操作符。

Table 9-39. JSON 操作符

操作符	右操作数的类型	描述	示例
->	int	获取JSON数组元素	'[1,2,3]'::json->2
->	text	获取JSON对象字段	'{"a":1,"b":2}'::json->'b'
->>	int	获取JSON数组元素为文本	'[1,2,3]'::json->>2
->>	text	获取JSON对象字段为文本	'{"a":1,"b":2}'::json->>'b'
#>	array of text	在指定的路径获取JSON对象	'{"a":[1,2,3],"b":[4,5,6]}'::json#>'{a,2}'
#>>	array of text	在指定的路径获取JSON对象为文本	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'

Table 9-40显示了可以用于创建和操作JSON （参阅Section 8.14）数据的函数。

Table 9-40. JSON 支持函数

函数	返回类型	描述
`array_to_json(anyarray [, pretty_bool])`	`json`	作为JSON返组。一个PostgreSQL数组成为一个JSON数组的组。 如果 `pretty_boo` true，将在维的元素之间添换行符。
`row_to_json(record [, pretty_bool])`	`json`	作为JSON返行。如果 `pretty_boo` true，将在第元素之间添加行符。
`to_json(anyelement)`	`json`	作为JSON返值。如果不是建的数组类型那么将会把这类型转换为 `json` 类型转换函数将用实现这个转换否则，除了数字，布尔值或值，其他的任值将用文本表示， 并且使用逸和双引号以其是合法的JS类型。
`json_array_length(json)`	`int`	返回最外层的JSON数组元数量。
`json_each(json)`	`SETOF key text, value json`	扩展最外层的JSON对象为键/值对。

key | value -----+------- a | "foo" b | "bar"

```
| `json_each_text(from_json json)` | `SETOF key text, value text` | 扩展最外层的JSON对象为一
```

key | value -----+------- a | foo b | bar

```
| `json_extract_path(from_json json, VARIADIC path_elems text[])` | `json` | 返回由`path_e
| `json_extract_path_text(from_json json, VARIADIC path_elems text[])` | `text` | 返回由`p
| `json_object_keys(json)` | `SETOF text` | 返回JSON对象中的一组键。只显示"外部"对象。 | `json_
```

json_object_keys

f1 f2

```
| `json_populate_record(base anyelement, from_json json, [, use_json_as_text bool=false]`
```

a | b ---+--- 1 | 2

```
| `json_populate_recordset(base anyelement, from_json json, [, use_json_as_text bool=fals
```

a | b ---+--- 1 | 2 3 | 4

```
| `json_array_elements(json)` | `SETOF json` | 扩展一个JSON数组到一组JSON元素的集合。 | `json_
```

value

1 true [2,false] ``` |

> **Note:** `json` 函数和操作符比输入函数类型可以实施更加严格的有效性需求。特别的，他们的检查更为紧密， 任何使用Unicode代理对到Unicode基本多文种平面以外的指定字符是正确的。
>
> **Note:** 这些函数和操作符中的许多将转换JSON文本中的Unicode逃逸到相应的UTF8字符（当数据库编码为UTF8时）。 在其他编码模式下，转义序列必须是ASCII字符，任何其他在Unicode逃逸序列中的代码点将导致一个错误。 通常，如果可能的话，最好避免JSON中Unicode逃逸和非UTF8数据库编码混合。
>
> **Note:** hstore扩展从 `hstore` 转换到 `json` ，所以转换了的 `hstore` 值作为JSON对象显示，而不是字符串值。

参阅Section 9.20获取有关聚集函数 `json_agg` 的信息， `json_agg` 有效的聚合记录值为JSON。

9.16. 序列操作函数

本节描述用于操作序列对象的函数， 也叫序列生成器或者就叫序列。序列对象都是用CREATE SEQUENCE 创建的特殊的单行表。一个序列对象通常用于为一个表的行生成唯一的标识符。 在Table 9-41 中列出的序列函数为我们从序列对象中获取后续的序列值提供了简单的、多用户安全的方法。

Table 9-41. 序列函数

函数	返回类型	描述
`currval(```regclass)	bigint	返回最近一次用 nextval 获取的指定序列的数值
lastval()	bigint	返回最近一次用 nextval 获取的任意序列的数值
`nextval(```regclass)	bigint	递增序列并返回新值
`setval(```regclass , bigint)	bigint	设置序列的当前数值
`setval(```regclass , bigint , boolean)	bigint	设置序列的当前数值以及 is_called 标志

被序列函数操作的序列是用 regclass 参数声明的，它只是序列在 pg_class 系统表里面的 OID 。不过， 你不需要手工查找 OID ， 因为 regclass 数据类型的输入转换器会帮你做这件事。只要写出单引号包围的序列名字即可， 因此它看上去像文本常量。要达到和处理普通SQL名字的兼容性， 这个字符串将转换成小写，除非在序列名字周围包含双引号，因此

```
nextval('foo')      _操作序列号 `foo`_
nextval('FOO')      _操作序列号`foo`_
nextval('"Foo"')    _操作序列号 `Foo`_
```

必要时序列名可以用模式修饰 :

```
nextval('myschema.foo')     _操作`myschema.foo`_
nextval('"myschema".foo')   _同上_
nextval('foo')              _在搜索路径中查找`foo`_
```

参阅Section 8.18获取有关 regclass 的更多信息。

> **Note:** 在PostgreSQL 8.1之前，序列函数的参数类型是 `text` 而不是 `regclass`，而上面描述的从文本字符串到 OID 值的转换将在每次调用的时候发生。为了向下兼容，这个机制仍然存在，但是在内部实际上是在函数调用前隐含地将 `text` 转换成 `regclass` 实现的。
>
> 如果你把一个序列函数的参数写成一个无修饰的文本字符串，那么它将变成类型为 `regclass` 的常量。因为这只是一个 OID，它将跟踪最初标识的序列，而不管后面是否改名、模式是否变化等等。这种"提前绑定"的行为通常是字段缺省和视图里面引用序列所需要的。但是有时候你可能想要"推迟绑定"，这个时候序列的引用是在运行时解析的。要获取推迟绑定的行为，我们可以强制存储为 `text` 常量，而不是 `regclass` 常量：
>
> ```
> nextval('foo'::text)      _`foo` 在运行时查找_
> ```
>
> 请注意，推迟绑定是PostgreSQL版本 8.1 之前唯一可用的行为，因此你可能需要在旧的应用里如此使用来保留旧有的语意。
>
> 当然，序列函数的参数也可以是表达式。如果它是一个文本表达式，那么隐含的转换将导致运行时的查找。

用的序列函数有：

`nextval`

递增序列对象到它的下一个数值并且返回该值。这个动作是自动完成的：即使多个会话并发运行 `nextval`，每个进程也会安全地收到一个唯一的序列值。

如果一个序列对象是带着缺省参数创建的，那么对它连续调用 `nextval` 将返回从1开始的后续的数值。其他的行为可以通过使用CREATE SEQUENCE 命令里的特殊参数获取；参考其命令参考页获取更多信息。

> **Important:** 为了避免从同一个序列获取数值的当前事务被阻塞，`nextval` 操作决不会回滚；也就是说，一旦一个数值已经被抓走，那么就认为它已经用过了，即使调用 `nextval` 的事务后面又退出了也一样。这就意味着退出的事务可能在序列赋予的数值中留下未使用的"空洞"。

`currval`

在当前会话中返回最近一次 `nextval` 抓到的该序列的数值。如果在本会话中从未在该序列上调用过 `nextval`，那么会报告一个错误。因为此函数返回一个会话范围的数值，它也能给出一个可预计的结果，可以判断其它会话是否执行过 `nextval` 函数。

`lastval`

返回当前会话里最近一次 `nextval` 返回的数值。这个函数等效于 `currval`，只是它不用序列名作为参数，它抓取当前会话里面最近一次 `nextval` 使用的序列的值。如果当前会话还没有调用过 `nextval`，那么调用 `lastval` 是会报错的。

`setval`

重置序列对象的计数器数值。2个参数的形式设置序列的 `last_value` 字段为声明数值并且将其 `is_called` 字段设置为 `true` ， 表示下一次 `nextval` 将在返回数值之前递增该序列。 `curval` 报告的值也设定为指定的值。在3个参数形式里 `is_called` 可以设置为 `true` 或 `false` 。设置为 `true` 和2参数的形式影响相同。 如果你把它设置为 `false` ， 那么下一次 `nextval` 将返回这里声明的数值而随后 `nextval` 才开始递增该序列。 因此在这种情况下 `currval` 报告的值没有改变（这是8.3之前版本的一个变化）。比如，

```
SELECT setval('foo', 42);           _下次`nextval`将返回43_
SELECT setval('foo', 42, true);     _和上面一样_
SELECT setval('foo', 42, false);    _下次`nextval`将返回42_
```

`setval` 返回的结果就是它的第二个参数的数值。

> **Important:** 因为序列是非事务性的，如果事务回滚了，由 `setval` 所做的改变也不会取消。

9.17. 条件表达式

本节描述在PostgreSQL里可用的SQL兼容的条件表达式。

> **Tip:** 如果你的需求超过这些条件表达式的能力，你可能会希望用一种更富表现力的编程语言写一个存储过程。

9.17.1. `CASE`

`CASE` 表达式是一种通用的条件表达式，类似于其它编程语言中的 if/else 语句。

```
CASE WHEN _condition_ THEN _result_
     [WHEN ...]
     [ELSE `_result_`]
END
```

`CASE` 子句可以用于任何表达式可以存在的地方。 `_condition_` 是一个返回 `boolean` 的表达式。如果条件的结果为真，那么 `CASE` 表达式的结果就是符合条件的 `_result_` ，并且不再处理剩余的 `CASE` 表达式。如果条件的结果为假，那么以相同方式搜寻任何随后的 `WHEN` 子句。如果没有 `WHEN` `_condition_` 为真， 那么表达式的结果就是在 `ELSE` 子句里的 `_result_` 。 如果省略了 `ELSE` 子句且没有匹配的条件，结果为 NULL 。

一个例子：

```
SELECT * FROM test;

 a
---
 1
 2
 3

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

所有 `_result_` 表达式的数据的类型都必须可以转换成单一的输出类型。 参阅Section 10.5获取更多细节。

下面这个"简单的" `CASE` 表达式是上面的通用形式的一个特殊的变种：

```
CASE _expression_
    WHEN _value_ THEN _result_
    [WHEN ...]
    [ELSE `_result_`]
END
```

先计算 `_expression_` 的值，然后与每个 `WHEN` 子句里声明的 `_value_` 表达式对比，直到找到一个相等的。 如果没有找到匹配的，则返回在 `ELSE` 子句里的 `_result_` (或者 NULL)。这个类似于 C 里的 `switch` 语句。

上面的例子可以用简单 `CASE` 语法来写：

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

`CASE` 表达式并不计算任何对于判断结果并不需要的子表达式。比如， 下面是一个可以避免被零除的方法：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **Note:** 正如在Section 35.6描述的那样，被 `IMMUTABLE` 标记的函数和操作符在计划查询时评估，而不是在执行时。 这意味着没有在查询执行时评估的子表达式的常量部分可能仍会在查询计划时评估。

9.17.2. `COALESCE`

```
COALESCE(_value_ [, ...])
```

`COALESCE` 返回它的第一个非 NULL 的参数值。如果所有参数都是null那么返回null。 它常用于在显示数据时用缺省值替换 NULL 。比如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

如果 `description` 非空那么返回它，否则如果 `short_description` 非空则返回它， 否则返回 `(none)` 。

和 `CASE` 表达式一样， `COALESCE` 只计算需要用来判断结果的参数；也就是说，在第一个非空参数右边的参数不会被计算。 这个符合 SQL 标准的函数提供了与某些其它数据库系统中的 `NVL` 和 `IFNULL` 类似的功能。

9.17.3. `NULLIF`

```
NULLIF(_value1_, _value2_)
```

当且仅当 `_value1_` 等于 `_value2_` 时， `NULLIF` 才返回 null 。否则它返回 `_value1_` 。 这些可以用于执行上面给出的 `COALESCE` 例子的反例：

```
SELECT NULLIF(value, '(none)') ...
```

在这个例子中，如果 `value` 是 `(none)` 那么返回 null，否则返回 `value` 。

9.17.4. `GREATEST` and `LEAST`

```
GREATEST(_value_ [, ...])
```

```
LEAST(_value_ [, ...])
```

`GREATEST` 和 `LEAST` 函数从一个任意数字表达式的列表里选取最大或者最小的数值。 这些表达式必须都可以转换成一个普通的数据类型，它将会是结果类型(参阅Section 10.5 获取细节)。列表中的 NULL 值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才会是 NULL 。

请注意 `GREATEST` 和 `LEAST` 都不是 SQL 标准，但却是很常见的扩展。 某些其他的数据库在任意一个参数为NULL时返回NULL，而不是所有参数都是NULL时。

9.18. 数组函数和操作符

Table 9-42显示了可以用于 array 类型的操作符。

Table 9-42. Array 操作符

操作符	描述	例子	
`=`	等于	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`	`t`
`<>`	不等于	`ARRAY[1,2,3] <> ARRAY[1,2,4]`	`t`
`<`	小于	`ARRAY[1,2,3] < ARRAY[1,2,4]`	`t`
`>`	大于	`ARRAY[1,4,3] > ARRAY[1,2,4]`	`t`
`<=`	小于或等于	`ARRAY[1,2,3] <= ARRAY[1,2,3]`	`t`
`>=`	大于或等于	`ARRAY[1,4,3] >= ARRAY[1,4,3]`	`t`
`@>`	包含	`ARRAY[1,4,3] @> ARRAY[3,1]`	`t`
`<@`	被包含于	`ARRAY[2,7] <@ ARRAY[1,7,4,2,6]`	`t`
`&&`	重叠(有共同元素)	`ARRAY[1,4,3] && ARRAY[2,1]`	`t`
	数组与		

`||`	数组连接	`ARRAY[1,2,3] || ARRAY[4,5,6]`	`{1,2,3,4,5`
`||`	数组与数组连接	`ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]`	`{{1,2,3},{`
`||`	元素与数组连接	`3 || ARRAY[4,5,6]`	`{3,4,5,6}`
`||`	数组与元素连接	`ARRAY[4,5,6] || 7`	`{4,5,6,7}`

数组比较是使用默认的 B-Tree 比较函数对所有元素逐一进行比较的。 多维数组的元素按照行顺序进行访问(最后的下标变化最快)。如果两个数组的内容相同但维数不等， 那么决定排序顺序的首要因素将是维数(原文：the first difference in the dimensionality information determines the sort order)， 这与PostgreSQL 8.2 之前的版本不同： 老版本认为内容相同的两个数组相等，即使它们的维数或下标范围并不相同。

参阅Section 8.15获取有关数组操作符行为的更多细节。 参阅Section 11.2获取哪个操作符支持索引操作的更多细节。

Table 9-43显示了可以用于数组类型的函数。 参阅Section 8.15获取更多信息以及使用这些函数的例子。

Table 9-43. Array 函数

函数	返回类型	描述
`array_append`(anyarray , anyelement`)	`anyarray`	向数组末尾添加元素
		连接

array_cat`( anyarray , anyarray`)	anyarray	两个数组	
array_ndims`( anyarray`)	int	返回数组的维数	
array_dims`( anyarray`)	text	返回数组维数的文本表示	
array_fill`( anyelement , int[] , [, int[]`])	anyarray	返回数组初始化提供的值和维度，可选下界不是1	
array_length`( anyarray , int`)	int	返回数组维度的长度	
array_lower`( anyarray , int`)	int	返回数组维数的下界	
array_prepend`( anyelement , anyarray`)	anyarray	向数组开头添加元素	
array_remove`( anyarray , anyelement`)	anyarray	从数组中删除所有等于给定值的元素（数	

		组必须是一维的)
array_replace`( anyarray , anyelement , anyelement`)	anyarray	用新值替换每个等于给定值的数组元素
array_to_string`( anyarray , text [, text`])	text	使用分隔符和null字符串连接数组元素
array_upper`( anyarray , int`)	int	返回数组维数的上界
string_to_array`( text , text [, text`])	text[]	使用指定的分隔符和null字符串把字符串分裂成数组元素
unnest`( anyarray`)	setof anyelement	扩大一个数组为一组行

在 `string_to_array` 中，如果分隔符参数是NULL， 输入字符串中的每个字符将在结果数组中变成一个独立的元素。如果分隔符是一个空白字符串， 那么整个输入字符串将变为一元素的数组。否则输入字符串将在每个分隔字符串处分裂。

在 `string_to_array` 中，如果省略null字符串参数或为NULL， 将没有输入字符串的子串被NULL代替。在 `array_to_string` 中， 如果省略null字符串参数或为NULL，在数组中的任何null元素将简单的跳过， 并且不再输出字符串中出现。

> **Note:** 在PostgreSQL版本9.1之前， `string_to_array` 有两个行为上的不同。 第一，当输入字符串长度为零时，它将返回一个空（零元素）数组而不是NULL。第二， 如果分隔字符串是NULL时，函数分隔输入为单独的字符，而不是和以前一样返回NULL。

也可以参阅Section 9.20获取关于 `array_agg` 聚集函数使用数组的信息。

{% raw %}

9.19. 范围函数和操作符

参阅Section 8.17获取范围类型的概述。

Table 9-44显示了范围类型可用的操作符。

Table 9-44. 范围操作符

操作符	描述	示例
=	等于	int4range(1,5) = '[1,4]'::int4range
<>	不等于	numrange(1.1,2.2) <> numrange(1.1,2.3)
<	小于	int4range(1,10) < int4range(2,3)
>	大于	int4range(1,10) > int4range(1,5)
<=	小于或等于	numrange(1.1,2.2) <= numrange(1.1,2.2)
>=	大于或等于	numrange(1.1,2.2) >= numrange(1.1,2.0)
@>	包含范围	int4range(2,4) @> int4range(2,3)
@>	包含元素	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp
<@	范围包含于	int4range(2,4) <@ int4range(1,7)
	元	

`<@`	素包含于	`42 <@ int4range(1,7)`
`&&`	重叠(有共同点)	`int8range(3,7) && int8range(4,12)`
`<<`	严格在左	`int8range(1,10) << int8range(100,110)`
`>>`	严格在右	`int8range(50,60) >> int8range(20,30)`
`&<`	没有延伸到右边	`int8range(1,20) &< int8range(18,20)`
`&>`	没有延伸到左边	`int8range(7,20) &> int8range(5,10)`
`-|-`	相邻	`numrange(1.1,2.2) -|- numrange(2.2,3.3)`
`+`	并集	`numrange(5,15) + numrange(10,20)`
`*`	交集	`int8range(5,15) * int8range(10,20)`
`-`	差集	`int8range(5,15) - int8range(10,20)`

简单的比较操作符 `<`， `>`， `<=` 和 `>=` 先比较下界，只有下界相等时才比较上界。 这种比较通常对范围不是很好用，但是为了在范围中允许构建B-tree索引才提供的。

左于/右于/邻近操作符当包含空范围时也会返回false；也就是， 不认为空范围在其他范围之前或之后。

并集和差集操作符在结果范围需要包含两个不相交的子范围时失败，因此不能表示这样一个范围。

Table 9-45显示了可以和范围一起使用的函数。

Table 9-45. 范围函数

函数	返回类型	描述	示例	结果
`lower`(anyrange)	范围元素类型	范围的下界	`lower(numrange(1.1,2.2))`	`1.1`
`upper`(anyrange)	范围元素类型	范围的上界	`upper(numrange(1.1,2.2))`	`2.2`
`isempty`(anyrange)	`boolean`	范围是空的?	`isempty(numrange(1.1,2.2))`	`false`
`lower_inc`(anyrange)	`boolean`	包涵下界?	`lower_inc(numrange(1.1,2.2))`	`true`
`upper_inc`(anyrange)	`boolean`	包含上界?	`upper_inc(numrange(1.1,2.2))`	`false`
`lower_inf`(anyrange)	`boolean`	下界无穷?	`lower_inf('(,)'::daterange)`	`true`
`upper_inf`(anyrange)	`boolean`	上界无穷?	`upper_inf('(,)'::daterange)`	`true`

如果范围是空或者需要的界限是无穷的， `lower` 和 `upper` 函数返回null。 `lower_inc` , `upper_inc` , `lower_inf` 和 `upper_inf` 函数均对空范围返回false。

9.20. 聚集函数

聚集函数从一组输入值里计算一个结果。 Table 9-46和Table 9-47 显示了内建的聚集函数。聚集函数的特殊语法在Section 4.2.7里解释。 请参考Section 2.7获取额外的介绍性信息。

Table 9-46. 通用聚集函数

函数	参数类型	返回类型
`array_agg(` *expression* `)`	任意	参数类型的数组
`avg(` *expression* `)`	`smallint` , `int` , `bigint` , `real` , `double precision` , `numeric` , or `interval`	对于任何整数类型输入，结果都是 `numeric` 类型。 对于任何浮点输入，结果都是 `double precision` 类型。 否则和输入数据类型相同。
`bit_and(` *expression* `)`	`smallint` , `int` , `bigint` , or `bit`	和参数数据类型相同
`bit_or(` *expression* `)`	`smallint` , `int` , `bigint` , or `bit`	和参数数据类型相同
`bool_and(` *expression* `)`	`bool`	`bool`
`bool_or(` *expression* `)`	`bool`	`bool`
`count(*)`	`bigint`	输入行数
`count(` *expression* `)`	任意	`bigint`
`every(` *expression* `)`	`bool`	`bool`
	`record`	`json`

`max(` *expression* `)`	任意数组、数值、字符串、日期/时间类型	和参数数据类型相同
`min(` *expression* `)`	任意数组、数值、字符串、日期/时间类型	和参数数据类型相同
`string_agg(` *expression* `,` *delimiter* `)`	(`text` , `text`) or (`bytea` , `bytea`)	和参数数据类型相同
`sum(` *expression* `)`	`smallint` , `int` , `bigint` , `real` , `double precision` , `numeric` , or `interval`	对于 `smallint` 或 `int` 输入，输出类型为 `bigint` 。 对于 `bigint` 输入，输出类型为 `numeric` ，对于浮点数输入， 输出类型为 `double precision` 。否则和输入数据类型相同。
`xmlagg(` *expression* `)`	`xml`	`xml`

请注意，除了 `count` 以外，这些函数在没有输入行时返回 NULL 。 尤其要指出的是 `sum` 函数在没有输入行时返回 NULL ， 而不是零。 `array_agg` 函数在没有输入行时返回null而不是空数组。 必要时可以用 `coalesce` 把 NULL 替换成零或空数组。

> **Note:** `bool_and` 和 `bool_or` 布尔聚集对应标准的 SQL 聚集 `every` 和 `any` 或 `some` 。 对于 `any` 和 `some` ，标准语法里面似乎有些内置的歧义：
>
> ```
> SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
> ```
>
> 这里的 `ANY` 既可以被认为是引出一个子查询， 也可以被认为是一个聚集(如果子查询返回布尔值的1行的话)。因此标准的名字无法用于这些聚集。
>
> **Note:** 习惯了其它 SQL 数据库管理系统的用户可能被用于全表计算的 `count` 的性能(之慢)惊住了。一个类似下面这样的查询：
>
> ```
> SELECT count(*) FROM sometable;
> ```
>
> 将需要努力与表的大小成正比：PostgreSQL 将需要扫面整个表或包含表中所有行的完整的索引。

聚集函数 `array_agg` ， `json_agg` ， `string_agg` 和 `xmlagg` ，以及类似用户定义的聚集函数，根据输入值的顺序产生意义不同的结果值。 这个顺序默认没有指定，但是可以通过在聚集函数调用时，写一个 `ORDER BY` 子句来控制， 就像Section 4.2.7描述的那样。另外， 通常可以从一个已排序的子查询中提供输入值。例如：

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

但此语法在SQL标准里不允许，不能移植到其它数据库系统。

Table 9-47展示了用于统计分析的聚集函数。 单独列出这些函数仅仅是为了避免和那些经常使用的聚集函数混在一起而已。 "描述"列中的 `_N_` 表示所有输入行中使得输入表达式不为 NULL 的行数。总的来说，如果计算本身变得没有意义， 那么返回值将是 NULL 。例如当 `_N_` 为零的时候。

Table 9-47. 统计聚集函数

函数	参数类型	返回类型	描
`corr(` *Y* , *X* `)`	`double precision`	`double precision`	相关系数
`covar_pop(` *Y* , *X* `)`	`double precision`	`double precision`	总体协方
`covar_samp(` *Y* , *X* `)`	`double precision`	`double precision`	样本协方
`regr_avgx(` *Y* , *X* `)`	`double precision`	`double precision`	自变量的 (`sum(``_`
`regr_avgy(` *Y* , *X* `)`	`double precision`	`double precision`	因变量的 (`sum(``_`
`regr_count(` *Y* , *X* `)`	`double precision`	`bigint`	两个表达 NULL 的
`regr_intercept(` *Y* , *X* `)`	`double precision`	`double precision`	根据所有 (_X_ , _ 最小二乘 一个线性 后返回该 轴截距
`regr_r2(` *Y* , *X* `)`	`double precision`	`double precision`	相关系数
`regr_slope(` *Y* , *X* `)`	`double precision`	`double precision`	根据所有 (_X_ , _ 最小二乘 一个线性 后返回该 率。
`regr_sxx(` *Y* , *X* `)`	`double precision`	`double precision`	`sum(``_X` sum(_X_ (自变量的

regr_sxy(*Y* , *X*)	double precision	double precision	sum(``_X *sum(_X_* sum(_Y_ (自变量和 的"乘方积
regr_syy(*Y* , *X*)	double precision	double precision	sum(``_Y sum(_Y_ (因变量的
stddev(*expression*)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	stddev_s 名(历史原
stddev_pop(*expression*)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	总体标准
stddev_samp(*expression*)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	样本标准
variance (_expression_)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	var_samp (历史原因
var_pop (_expression_)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	总体方差 差的平方
var_samp (_expression_)	smallint , int , bigint , real , double precision , 或 numeric	对于浮点类型的输入返回 double precision ，其他输入返回 numeric	样本方差 差的平方

9.21. 窗口函数

窗口函数提供跨行相关的当前查询行集执行计算的能力。 参阅Section 3.5获取这个特性的介绍。

Table 9-48列出了内建的窗口函数。 注意必须使用窗口函数的语法调用这些函数；一个 `OVER` 子句是必需的。

除了这些函数外，任何内建的或用户定义的聚集函数都可以作为窗口函数（见 Section 9.20关于内建聚集函数的列表）。 仅当调用跟着 `OVER` 子句的聚集函数，作为窗口函数；否则它们作为常规的聚合。

Table 9-48. 通用窗口函数

函数	返回类型	描述
`row_number()`	`bigint`	在其分区中的当前行号，从1计
`rank()`	`bigint`	有间隔的当前行排名；与它的第一个相同行的 `row_number` 相同
`dense_rank()`	`bigint`	没有间隔的当前行排名；这个函数计数对等组。
`percent_rank()`	`double precision`	当前行的相对排名: (`rank` - 1) / (总行数 - 1)
`cume_dist()`	`double precision`	当前行的相对排名：(前面的行数或与当前行相同的行数)/(总行数)
`ntile(` *num_buckets* `` `integer ``)`	`integer`	从1到参数值的整数范围，尽可能相等的划分分区。
`lag(` *value* `` `any `` [, `_offset_` `integer` [, `_default_` `any`]])`	类型同 `_value_`	计算分区当前行的前 `_offset_` 行，返回 `_value_` 。如果没有这样的行，返回 `_default_` 替代。 `_offset_` 和 `_default_` 都是当前行计算的结果。如果忽略了，则 `_offset_` 默认是1， `_default_` 默认是null。
`lead(` *value* `` `any `` [, `_offset_` `integer` [, `_default_` `any`]])`	类型同``_value_	计算分区当前行的后 `_offset_` 行，返回 `_value_` 。如果没有这样的行，返回 `_default_` 替代。 `_offset_` 和 `_default_` 都是当前行计算的结果。如果忽略了，则 `_offset_` 默认是1， `_default_` 默认是null。
`first_value(` *value* `` `any ``)`	类型同``_value_	返回窗口第一行的计算 `_value_` 值。
`last_value(` *value* `` `any ``)`	类型同``_value_	返回窗口最后一行的计算 `_value_` 值。
`nth_value(` *value* `` `any `` , `_nth_` `integer`)`	类型同``_value_	返回窗口第 `_nth_` 行的计算 `_value_` 值（行从1计数）；没有这样的行则返回 null。

在Table 9-48列出的所有函数，依赖于与窗口定义有关的 `ORDER BY` 子句指定的排序。同行是说在 `ORDER BY` 排序时不唯一的行。定义的这四个排名函数，对于任何两个同行的答案相同。

注意 `first_value` ， `last_value` ， 和 `nth_value` 只考虑"window frame"内的行，其默认情况下，包含从分区的开始行直到当前行的最后同行。 像 `last_value` 和 `nth_value` 有时会给出没有用的结果。 您可以通过向 `OVER` 子句添加合适的框架规范（ `RANGE` 或者 `ROWS` ） 来重新定义该框架。参阅Section 4.2.8获取框架定义的信息。

当一个聚集函数作为窗口函数使用时，将聚合超过当前行的窗框内的行。 一个使用 `ORDER BY` 和默认窗框定义处理"运行时求和"类型的行为的聚集函数， 可能不是想要的结果。为了获取超过整个分区聚合，忽略 `ORDER BY` 或者使用 `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` 。 其它窗框规格可以用来获取其它的效果。

> **Note:** SQL 标准为 `lead` , `lag` , `first_value` , `last_value` , 和 `nth_value` 定义了一个 `RESPECT NULLS` 或 `IGNORE NULLS` 选项。 这个在PostgreSQL没有实现：行为总是与标准默认相同，即 `RESPECT NULLS` 。 同样用于 `nth_value` 的标准 `FROM FIRST` 或 `FROM LAST` 选项也没有实现：只支持默认 `FROM FIRST` 行为。 （您可以通过 `ORDER BY` 排序取反获取到 `FROM LAST` 的结果。）

9.22. 子查询表达式

本节描述PostgreSQL里面与SQL 兼容的子查询表达式。所有本节中的表达式都返回布尔值(真/假)结果。

9.22.1. `EXISTS`

```
EXISTS (_subquery_)
```

`EXISTS` 的参数是一个任意的 `SELECT` 语句，或者说子查询。 系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么 `EXISTS` 的结果就为 "真"；如果子查询没有返回任何行，那么 `EXISTS` 的结果是"假"。

子查询可以引用包围它的查询的变量，这些变量在该子查询的每一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以生成至少一行为止，而不是等到全部结束。 在这里写有副作用的子查询是不明智的(比如调用序列函数)；这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容， 所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用下面的形式写 `EXISTS` 测试：`EXISTS(SELECT 1 WHERE ...)`。 不过这条规则也有例外，比如那些使用 `INTERSECT` 的子查询。

下面这个简单的例子类似在 `col2` 上的一次内连接， 但是它为每个 `tab1` 的行生成最多一个输出， 即使存在多个匹配 `tab2` 的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. `IN`

```
_expression_ IN (_subquery_)
```

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。 左边表达式对子查询结果的每一行进行一次计算和比较。如果找到任何相等的子查询行， 则 `IN` 结果为"真"。如果没有找到任何相等行，则结果为"假" (包括子查询没有返回任何行的情况)。

请注意，如果左边表达式的值为 NULL ，或者没有相等的右边值并且至少有一个右边行生成 NULL ， 那么 `IN` 的结果将是 NULL ，而不是假。这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

和 `EXISTS` 一样，假设子查询将被完全运行是不明智的。

```
_row_constructor_ IN (_subquery_)
```

左边是一个行构造器(如Section 4.2.13所述)，右边是一个圆括弧括起来的子查询， 它必须返回和左边行构造器一样多的字段。左边表达式对子查询结果的每一行进行一次计算和比较。 如果找到相等的子查询行，则 `IN` 结果为"真"。如果没有找到任何相等行， 则结果为"假"(包括子查询没有返回任何行的情况)。

表达式或子查询行里的 NULL 遵照 SQL 处理布尔值和 NULL 组合时的规则。 如果两个行对应的字段都相等且非空，那么这两行相等；如果任意对应字段不等且非空， 那么这两行不等；否则结果是未知(NULL)。如果每一行的结果都是不等或 NULL ， 并且至少有一个 NULL ，那么 `IN` 的结果是 NULL 。

9.22.3. `NOT IN`

```
_expression_ NOT IN (_subquery_)
```

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。 左边表达式对子查询结果的每一行进行一次计算和比较。如果只找到不相等的子查询行 (包括子查询没有返回任何行的情况)，则 `NOT IN` 结果为"真"。 如果找到任何相等行，则结果为"假"。

请注意，如果左边表达式的值为 NULL ，或者没有相等的右边值并且至少有一个右边行生成 NULL ， 那么 `NOT IN` 的结果将是 NULL ，而不是真。这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

和 `EXISTS` 一样，假设子查询将被完全运行是不明智的。

```
_row_constructor_ NOT IN (_subquery_)
```

左边是一个行构造器(如Section 4.2.13所述)， 右边是一个圆括弧括起来的子查询，它必须返回和左边行构造器一样多的字段。 左边表达式对子查询结果的每一行进行一次计算和比较。 如果只出现不相等的子查询行， 则 `NOT IN` 结果为"真"。(包括子查询没有返回任何行的情况)。 如果找到相等的子查询行，则结果为"假"。

表达式或子查询行里的 NULL 遵照 SQL 处理布尔值和 NULL 组合时的规则。 如果两个行对应的字段都相等且非空，那么这两行相等；如果任意对应字段不等且非空， 那么这两行不等；否则结果是未知(NULL)。如果每一行的结果都是不等或 NULL ， 并且至少有一个 NULL

，那么 `NOT IN` 的结果是 NULL 。

9.22.4. `ANY` / `SOME`

```
_expression_ _operator_ ANY (_subquery_)
_expression_ _operator_ SOME (_subquery_)
```

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。 左边表达式使用 `_operator_` 对子查询结果的每一行进行一次计算和比较， 其结果必须是布尔值。如果至少获得一个真值，则 `ANY` 结果为"真"。 如果全部获得假值，则结果是"假"(包括子查询没有返回任何行的情况)。

`SOME` 是 `ANY` 的同意词。 `IN` 等效于 `= ANY` 。

请注意，如果没有获得任何真值并且至少有一个右边行在该操作符上生成 NULL ， 那么 `ANY` 的结果将是 NULL ， 而不是假。 这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

和 `EXISTS` 一样，假设子查询将被完全运行是不明智的。

```
_row_constructor_ _operator_ ANY (_subquery_)
_row_constructor_ _operator_ SOME (_subquery_)
```

左边是一个行构造器(如Section 4.2.13所述)， 右边是一个圆括弧括起来的子查询，它必须返回和左边行构造器一样多的字段。 左边表达式使用 `_operator_` 对子查询结果的每一行进行一次计算和比较。 如果至少获得一个真值，则 `ANY` 结果为"真"。 如果全部获得假值，则结果是"假"(包括子查询没有返回任何行的情况)。 如果没有获得任何真值并且至少有一个行返回 NULL ，那么结果将是 NULL。

查看Section 9.23.5获取关于逐行比较的细节。

9.22.5. `ALL`

```
_expression_ _operator_ ALL (_subquery_)
```

右边是一个圆括弧括起来的子查询，它必须只返回一个字段。 左边表达式使用 `_operator_` 对子查询结果的每一行进行一次计算和比较， 其结果必须是布尔值。如果全部获得真值， `ALL` 结果为"真" (包括子查询没有返回任何行的情况)。如果至少获得一个假值，则结果是"假"。 如果比较不会返回任何假值，并且至少一个行返回 NULL，则结果为 NULL。

`NOT IN` 等效于 `<> ALL` 。

和 `EXISTS` 一样，假设子查询将被完全运行是不明智的。

```
_row_constructor_ _operator_ ALL (_subquery_)
```

左边是一个行构造器(如Section 4.2.13所述)， 右边是一个圆括弧括起来的子查询，它必须返回和左边行构造器一样多的字段。 左边表达式使用 `_operator_` 对子查询结果的每一行进行一次计算和比较。 如果全部获得真值， `ALL` 结果为"真" (包括子查询没有返回任何行的情况)。如果至少获得一个假值，则结果是"假"。 如果比较不会返回任何假值，并且至少一个行返回 NULL，则结果为 NULL。

查看Section 9.23.5以获取关于逐行比较的细节。

9.22.6. 逐行比较

```
_row_constructor_ _operator_ (_subquery_)
```

左边是一个行构造器(如Section 4.2.13所述)， 右边是一个圆括弧括起来的子查询，它必须返回和左边行构造器一样多的字段。 而且，该子查询不能返回超过 1 行结果(返回零行相当于 NULL)。 左边表达式对子查询的唯一结果行进行计算和比较。

查看Section 9.23.5以获取关于逐行比较的细节。

9.23. 行和数组比较

本节描述几个特殊的构造，用于在多组值之间进行多重比较。 这些形式语法上和上一节的子查询形式相关，但是不涉及子查询。 这种形式涉及的数组子表达式是PostgreSQL的扩展；其它的是SQL兼容的。所有本节记录的表达式形式都返回布尔值(真/假)。

9.23.1. `IN`

```
_expression_ IN (_value_ [, ...])
```

右边是一个圆括弧包围的标量列表。如果左边的表达式结果等于任何右边表达式中的一个，结果为"真"。它是下面这种方式的缩写

```
_expression_ = _value1_
OR
_expression_ = _value2_
OR
...
```

请注意，如果左边表达式的值为 NULL ，或者没有相等的右边值并且至少有一个右边表达式的值为 NULL ， 那么 `IN` 的结果将是 NULL ，而不是假。这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

9.23.2. `NOT IN`

```
_expression_ NOT IN (_value_ [, ...])
```

右边是一个圆括弧包围的标量列表。如果左边的表达式结果不等于任何右边表达式，结果为"真"。 它是下面这种方式的缩写

```
_expression_ <> _value1_
AND
_expression_ <> _value2_
AND
...
```

请注意，如果左边表达式的值为 NULL ，或者没有相等的右边值并且至少有一个右边表达式的值为 NULL ， 那么 `NOT IN` 的结果将是 NULL ，而不是真。这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

> **Tip:** `x NOT IN y` 在所有场合都等价于 `NOT (x IN y)`。 但是，在处理 NULL 的时候，用 `NOT IN` 比用 `IN` 更容易迷惑新手。 最好用正逻辑来表达你的条件。

9.23.3. `ANY` / `SOME` (array)

```
_expression_ _operator_ ANY (_array expression_)
_expression_ _operator_ SOME (_array expression_)
```

右边是一个圆括弧包围的表达式，它必须生成一个数组值。左边表达式使用 `_operator_` 对数组的每一个元素进行一次计算和比较，其结果必须是布尔值。如果至少获得一个真值，则 `ANY` 结果为"真"。如果全部获得假值，则结果是"假" (包括数组不含任何元素的情况)。

如果数组表达式的值为 NULL ，那么 `ANY` 的结果也为 NULL 。 如果左边表达式的值为 NULL ，那么 `ANY` 的结果通常也为 NULL (某些不严格的比较操作符可能得到不同的结果)。另外，如果右边的数组表达式中包含 NULL 元素并且没有为真的比较结果， 那么 `ANY` 的结果将是 NULL(某些不严格的比较操作符可能得到不同的结果)， 而不是假。这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

`SOME` 是 `ANY` 的同意词。

9.23.4. `ALL` (array)

```
_expression_ _operator_ ALL (_array expression_)
```

右边是一个圆括弧包围的表达式，它必须生成一个数组值。左边表达式使用 `_operator_` 对数组的每一个元素进行一次计算和比较，其结果必须是布尔值。如果全部获得真值， `ALL` 结果为"真"(包括数组不含任何元素的情况)。如果至少获得一个假值，则结果是"假"。

如果数组表达式的值为 NULL ，那么 `ALL` 的结果也为 NULL 。 如果左边表达式的值为 NULL ，那么 `ALL` 的结果通常也为 NULL (某些不严格的比较操作符可能得到不同的结果)。另外，如果右边的数组表达式中包含 NULL 元素并且没有为假的比较结果，那么 `ALL` 的结果将是 NULL(某些不严格的比较操作符可能得到不同的结果)，而不是真。 这个行为遵照 SQL 处理布尔值和 NULL 组合时的规则。

9.23.5. 逐行比较

```
_row_constructor_ _operator_ _row_constructor_
```

两边都是一个Section 4.2.13所述的行构造器； 两个行的字段数必须相同。两边都被计算并且逐行比较。目前，用于比较的 `_operator_` 操作符仅允许为 `=`，`<>`，`<`，`<=`，`>`，`>=` 或与其具有相似的语意。 特别地，如果一个操作符属于 B-tree 操作符类，那么该操作符可以是一个行比较操作符或除 `=` 之外的 B-tree 操作符类。

`=` 和 `<>` 与其它操作符稍有区别。如果两行对应的元素全都非空且相等， 那么这两行就被认为是相等的；如果两行对应的元素中有任意一对非空且不等， 那么这两行就被认为是不等的；否则这两行的比较结果是未知(NULL)。

对于 `<`，`<=`，`>`，`>=` 操作符， 行中的元素将按照从左到右的顺序依次进行比较， 直到遇见一对不相等的元素或者一对 NULL 值。 如果这对元素中存在至少一个 NULL 值，那么比较的结果是 NULL ； 否则这对元素的比较结果就是最终的比较结果。例如， `ROW(1,2,NULL) < ROW(1,3,0)` 的结果是真而不是 NULL，因为比较到第二对元素的时候就已经得到了最终结果，不需要对第三对元素进行比较了。

> **Note:** 在PostgreSQL 8.2之前， `<`，`<=`，`>`，`>=` 并不遵守 SQL 标准。比如， `ROW(a,b) < ROW(c,d)` 将等价于 `a < c AND b < d`， 而正确的做法应当是等价于 `a < c OR (a = c AND b < d)`。

```
_row_constructor_ IS DISTINCT FROM _row_constructor_
```

这个构造类似于 `<>` 行比较，但是它对 NULL 输入不生成 NULL， 而是认为任何 NULL 都不等于任何非 NULL，并且 NULL 之间是相等的。因此， 结果要么是真要么是假，而绝不会是未知(NULL)。

```
_row_constructor_ IS NOT DISTINCT FROM _row_constructor_
```

这个构造类似于 `=` 行比较，但是它对 NULL 输入不生成 NULL， 而是认为任何 NULL 都不等于任何非 NULL，并且 NULL 之间是相等的。 因此，结果要么是真要么是假，而绝不会是未知(NULL)。

> **Note:** 如果结果依赖于比较两个NULL值或一个NULL值和一个非NULL值，SQL 规范要求逐行比较返回 NULL。 PostgreSQL仅当比较两个行构造的结果或一个行构造器和子查询的输出时这样做 (就像Section 9.22描述的那样)。在其它情况下， 两个复合类型的值进行比较， 认为两个NULL字段值是相等的，并且NULL大于非NULL。这是必要的，如此才能有一致的排序和复合类型的索引行为。

9.24. 返回集合的函数

本节描述那些可能返回多于一行的函数。在这个类中最广泛使用的函数是序列号生成函数，如Table 9-49和Table 9-50所述。 另外，更专业的集合返回函数在这个手册的其他地方描述。

Table 9-49. 序列号生成函数

函数	参数类型	返回类型
`generate_series(``start`, `stop`)	`int` 或 `bigint`	`setof int` 或 `setof bigint` (与参数类型相同)
`generate_series(``start` , `stop` , `step`)	`int` 或 `bigint`	`setof int` 或 `setof bigint` (与参数类型相同)
`generate_series(``start`, `stop`, `step` `interval`)	`timestamp` 或 `timestamp with time zone`	`setof timestamp` 或 `setof timestamp with time z` (与参数类型相同)

如果 `step` 是正数且 `start` 大于 `stop`， 那么返回零行。相反，如果 `step` 是负数且 `start` 小于 `stop`， 那么也返回零行。如果输入是 `NULL`， 同样产生零行。 `step` 为零则是一个错误。下面是一些例子：

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

-- this example relies on the date-plus-integer operator
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

Table 9-50. 下标生成函数

函数	返回类型	描述
`` `generate_subscripts(```array anyarray `` , `dim int`)	`setof int`	生成一系列包括给定数组的下标。
`` `generate_subscripts(```array anyarray `` , `dim int` , `reverse boolean`)	`setof int`	生成一系列包括给定数组的下标。当 `reverse` 为真时，该系列则以相反的顺序返回。

`generate_subscripts` 是一个为给定数组中的指定维度生成有效下标集的便利函数。 如果数组中没有所请求的维度或者NULL数组，返回0行（但是会给数组元素为空的返回有效下标）。下面一些例子：

```
-- basic usage
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- presenting an array, the subscript and the subscripted
-- value requires a subquery
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- unnest a 2D array
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
postgres=# SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

9.25. 系统信息函数

Table 9-51显示了几个抽取会话及系统信息的函数。

另外在本节所列出的函数，有一些函数有关统计系统，也提供系统信息。 参阅Section 27.2.2获取更多信息。

Table 9-51. 会话信息函数

Name	Return Type	Description
`current_catalog`	`name`	当前数据库名（在SQL标准里叫"catalog"）
`current_database()`	`name`	当前数据库名
`current_query()`	`text`	当前执行的查询文本，由客户端提交（可能包含多于1句）
`current_schema [()]`	`name`	当前模式名
`current_schemas(boolean)`	`name[]`	搜索路径中的模式名字，包括可选的隐式模式
`current_user`	`name`	当前执行环境下的用户名
`inet_client_addr()`	`inet`	连接的远端地址
`inet_client_port()`	`int`	连接的远端端口
`inet_server_addr()`	`inet`	连接的本地地址
`inet_server_port()`	`int`	连接的本地端口
`pg_backend_pid()`	`int`	连接到当前会话的服务器进程 ID
`pg_conf_load_time()`	`timestamp with time zone`	配置加载时间
`pg_is_other_temp_schema(oid)`	`boolean`	是否为另一个会话的临时模式?
`pg_listening_channels()`	`setof text`	正在侦听的当前会话的信道名称
`pg_my_temp_schema()`	`oid`	会话的临时模式的OID，不存在则为 0
`pg_postmaster_start_time()`	`timestamp with time zone`	服务器启动时间

`pg_trigger_depth()`	`int`	PostgreSQL 触发器的当前嵌套级别（如果没有直接或间接的从一个触发器内部调用，那么是0）
`session_user`	`name`	会话用户名
`user`	`name`	等价于 `current_user`
`version()`	`text`	PostgreSQL 版本信息

> **Note:** `current_catalog` , `current_schema` , `current_user` , `session_user` , 和 `user` 在SQL里有特殊的语法： 调用他们时结尾不能跟圆括号。 （在PostgreSQL， `current_schema` 可选的可以有括号，但是其它的不能。）

`session_user` 通常是连接当前数据库的初始用户， 不过超级用户可以用SET SESSION AUTHORIZATION修改这个设置。 `current_user` 是用于权限检查的用户标识。通常，它总是等于会话用户， 但是可以通过SET ROLE改变它。 在函数执行的过程中随着属性 `SECURITY DEFINER` 的改变，其值也会改变。 用 Unix 术语来说，会话用户是"真实用户"，而当前用户是"有效用户"。

`current_schema` 返回在搜索路径前端的模式名字(如果搜索路径为空则返回 NULL)。 如果创建表或者其它命名对象时没有声明目标模式，那么它将是用于这些对象的模式。 `current_schemas(boolean)` 返回一个搜索路径中所有模式名字的数组。 布尔选项决定像 `pg_catalog` 这样隐含包含的系统模式是否包含在返回的搜索路径中。

> **Note:** 搜索路径可以通过运行时设置更改。命令是：
>
> ```
> SET search_path TO _schema_ [, `_schema_`, ...]
> ```

`pg_listening_channels` 返回当前会话正在监听的一组信道名称。 见LISTEN获取更多信息。

`inet_client_addr` 返回当前客户端的IP地址， 而 `inet_client_port` 则返回当前客户端的端口号。 `inet_server_addr` 返回服务器接收当前连接用的 IP 地址， 而 `inet_server_port` 返回接收当前连接的端口号。 如果是通过 Unix-domain socket 连接的，那么所有这些函数都返回 NULL 。

`pg_my_temp_schema` 返回当前会话的临时模式 OID ， 如果不存在的话则返回 0(因为没有创建任何临时表)。 `pg_is_other_temp_schema` 返回给定的 OID 是否为其它会话的临时模式 OID ，这个函数是有实用价值的， 比如，在显示一个目录的时候排除掉其它会话的临时表。

`pg_postmaster_start_time` 返回服务器启动时的 `timestamp with time zone` 。

`pg_conf_load_time` 返回最后加载服务器配置文件的时间戳。 （如果当前会话在那时还活动，将是当前会话本身重新读取配置文件的时间， 所以读取的时间会在不同的会话中稍微有所不同。否则， 它是postmaster进程重新读取配置文件的时间。）

`version` 返回一个描述PostgreSQL服务器版本信息的字符串。

Table 9-52列出那些允许用户在程序里查询对象访问权限的函数。 参阅Section 5.6获取更多有关权限的信息。

Table 9-52. 访问权限查询函数

名字	返回类型	描述
`has_any_column_privilege`(user , table , privilege`)	`boolean`	指定用户是否有访问表任何列的权限
`has_any_column_privilege`(table , privilege`)	`boolean`	当前用户是否有访问表任何列的权限
`has_column_privilege`(user , table , column , privilege`)	`boolean`	指定用户是否有访问列的权限
`has_column_privilege`(table , column , privilege`)	`boolean`	当前用户是否有访问列的权限
`has_database_privilege`(user , database , privilege`)	`boolean`	指定用户是否有访问数据库的权限
`has_database_privilege`(database , privilege`)	`boolean`	当前用户是否有访问数据库的权限
`has_foreign_data_wrapper_privilege`(user , fdw , privilege`)	`boolean`	指定用户是否有访问外部数据封装器的权限
`has_foreign_data_wrapper_privilege`(fdw , privilege`)	`boolean`	当前用户是否有访问外部数据封装器的权限

`has_function_privilege`(user , function , privilege`)	`boolean`	指定用户是否有访问函数的权限
`has_function_privilege`(function , privilege`)	`boolean`	当前用户是否有访问函数的权限
`has_language_privilege`(user , language , privilege`)	`boolean`	指定用户是否有访问语言的权限
`has_language_privilege`(language , privilege`)	`boolean`	当前用户是否有访问语言的权限
`has_schema_privilege`(user , schema , privilege`)	`boolean`	指定用户是否有访问模式的权限
`has_schema_privilege`(schema , privilege`)	`boolean`	当前用户是否有访问模式的权限
`has_sequence_privilege`(user , sequence , privilege`)	`boolean`	指定用户是否有访问序列的权限
`has_sequence_privilege`(sequence , privilege`)	`boolean`	当前用户是否有访问序列的权限
`has_server_privilege`(user , server , privilege`)	`boolean`	指定用户是否有访问外部服务的权限
`has_server_privilege`(server , privilege`)	`boolean`	当前用户是否有访问外部服务的权限
`has_table_privilege`(user , table , privilege`)	`boolean`	指定用户是否有访问表的权限
		当前用户

`has_table_privilege`(table , privilege`)	`boolean`	是否有访问表的权限
`has_tablespace_privilege`(user , tablespace , privilege`)	`boolean`	指定用户是否有访问表空间的权限
`has_tablespace_privilege`(tablespace , privilege`)	`boolean`	当前用户是否有访问表空间的权限
`pg_has_role`(user , role , privilege`)	`boolean`	指定用户是否有角色的权限
`pg_has_role`(role , privilege`)	`boolean`	当前用户是否有角色的权限

`has_table_privilege` 检查用户是否可以用特定的方式访问表。 用户可以通过名字或OID（pg_authid.oid）来指定， `public` 表明PUBLIC伪角色， 或如果缺省该参数，则使用 `current_user` 。该表可以通过名字或者 OID 声明。 因此，实际上有六种 `has_table_privilege` 变体， 我们可以通过它们的参数数目和类型来区分它们。如果用名字声明，那么在必要时可以用模式进行修饰。 所希望的权限类型是用一个文本字符串来声明的，必须是 `SELECT` ， `INSERT` ， `UPDATE` ， `DELETE` ， `TRUNCATE` ， `REFERENCES` 或 `TRIGGER` 之一。 可选， 可以添加 `WITH GRANT OPTION` 到权限类型， 以测试权限是否拥有授权选项。也可以用逗号分隔多个列出的权限类型，如果拥有任何所列出的权限， 则结果便为 `true` 。 （权限字符串不区分大小写， 权限名之间允许有额外空白但不属于权限名的部分。）一些例子：

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

`has_sequence_privilege` 检查用户是否可以用特定的方式访问序列。 参数可能与 `has_table_privilege` 类似。 想要的访问权限必须为 `USAGE` ， `SELECT` 或 `UPDATE` 之一。

`has_any_column_privilege` 检查用户是否可以用特定的方式访问表的任何列。 其参数可能与 `has_table_privilege` 类似，除了想要的权限类型必须是 `SELECT` ， `INSERT` ， `UPDATE` ，或 `REFERENCES` 的一些组合。 请注意，在表级别拥有任何这些权限隐含授予它为每个表列，因此如果与 `has_table_privilege` 参数相同， `has_any_column_privilege` 总是返回 `true` 。 但是如果有至少一列的列级权限授予也成功。

`has_column_privilege` 检查用户是否可以用特定的方式访问一列。 其可能的参数类似于 `has_table_privilege` ， 可以通过列名或属性数添加列。想要的访问权限类型必须是 `SELECT` ， `INSERT` ， `UPDATE` ， 或 `REFERENCES` 的一些组合。 请注意，在表级别拥有任何这些权限隐含授予它为每个表列。

`has_database_privilege` 检查一个用户是否能以特定方式访问一个数据库。 它可能的参数类似 `has_table_privilege` 。权限类型必须是 `CREATE` ， `CONNECT` ， `TEMPORARY` ， `TEMP` (等价于 `TEMPORARY`)的一些组合。

`has_function_privilege` 检查一个用户是否能以特定方式访问一个函数。 它可能的参数类似 `has_table_privilege` 。 我们声明一个函数用的是文本字符串而不是 OID，允许的输入和 `regprocedure` 数据类型一样(参阅Section 8.18)。权限类型必须是 `EXECUTE` 。 一个例子如下：

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

`has_foreign_data_wrapper_privilege` 检查一个用户是否能以特定的方式访问外部数据封装器。 它的可能的参数类似 `has_table_privilege` 。权限类型必须是 `USAGE` 。

`has_language_privilege` 检查一个用户是否能以特定方式访问一个过程语言。 它可能的参数类似 `has_table_privilege` 。权限类型必须是 `USAGE` 。

`has_schema_privilege` 检查一个用户是否能以特定方式访问一个模式。 它可能的参数类似 `has_table_privilege` 。权限类型必须是 `CREATE` 或 `USAGE` 的一些组合。

`has_server_privilege` 检查一个用户是否能以特定方式访问一个外部服务器。 它可能的参数类似 `has_table_privilege` 。权限类型必须是 `USAGE` 。

`has_tablespace_privilege` 检查一个用户是否能以特定方式访问一个表空间。 它可能的参数类似 `has_table_privilege` 。权限类型必须是 `CREATE` 。

`pg_has_role` 检查一个用户是否能以特定方式访问一个角色。 它可能的参数类似 `has_table_privilege` ，除了 `public` 不能用做用户名。权限类型必须是 `MEMBER` 或 `USAGE` 的一些组合。 `MEMBER` 表示的是角色中的直接或间接成员关系(也就是 `SET ROLE` 的权限)，而 `USAGE` 表示角色的权限是否无需 `SET ROLE` 即可立即生效。

Table 9-53 显示了那些判断一个对象是否在当前模式搜索路径中可见的函数。 比如，如果一个表所在的模式在搜索路径中，并且没有同名的表出现在搜索路径的更靠前的地方， 那么就说这个表是可见的。它等效于表可以不带明确模式修饰进行引用。比如，要列出所有可见表的名字：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Table 9-53. 模式可见性查询函数

名字	返回类型	描述
`pg_collation_is_visible(```collation_oid)	boolean	该排序是否在搜索路径中可见
`pg_conversion_is_visible(```conversion_oid)	boolean	该转换是否在搜索路径中可见
`pg_function_is_visible(```function_oid)	boolean	该函数是否在搜索路径中可见
`pg_opclass_is_visible(```opclass_oid)	boolean	该操作符类是否在搜索路径中可见
`pg_operator_is_visible(```operator_oid)	boolean	该操作符是否在搜索路径中可见
`pg_opfamily_is_visible(```opclass_oid)	boolean	该操作符族是否在搜索路径中可见
`pg_table_is_visible(```table_oid)	boolean	该表是否在搜索路径中可见
`pg_ts_config_is_visible(```config_oid)	boolean	该文本检索配置是否在搜索路径中可见
`pg_ts_dict_is_visible(```dict_oid)	boolean	该文本检索词典是否在搜索路径中可见
`pg_ts_parser_is_visible(```parser_oid)	boolean	该文本搜索解析是否在搜索路径中可见
`pg_ts_template_is_visible(```template_oid)	boolean	该文本检索模板是否在搜索路径中可见
`pg_type_is_visible(```type_oid)	boolean	该类型（或域）是否在搜索路径中可见

每个函数执行一种数据库对象类型的可见性检查。请注意 `pg_table_is_visible` 还可用于视图、索引、序列。 `pg_type_is_visible` 还可用于域。 对于函数和操作符，如果在搜索路径中没有名字相同并且参数的数据类型 也相同的对象出现在路径中更靠前的位置，那么该对象就是可见的。对于操作符类， 则要同时考虑名字和相关的索引访问方法。

所有这些函数都需要使用 OID 来标识要被检查的对象。如果你想通过名字测试对象， 那么使用 OID 别名类型(`regclass` , `regtype` , `regprocedure` , `regoperator` , `regconfig` 或 `regdictionary`)将会很方便。例如：

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

请注意用这种方法测试一个未经修饰的名字没什么意义，因为如果一个名字可以被识别， 那它首先必须是可见的。

Table 9-54列出了从系统表中抽取信息的函数。

Table 9-54. 系统表信息函数

名字	返回类型	
`format_type(```type_oid , typemod)	text	获取一个数据类型
`pg_describe_object(```catalog_id , object_id , object_sub_id)	text	获取一个数据库对
`pg_identify_object(```catalog_id oid , object_id oid , object_sub_id integer)	type text , schema text , name text , identity text	获取一个数据库对
`pg_get_constraintdef(```constraint_oid)	text	获取一个约束的定
`pg_get_constraintdef(```constraint_oid , pretty_bool)	text	获取一个约束的定
`pg_get_expr(```pg_node_tree , relation_oid)	text	反编译一个表达式 的任何 Var 都引用
`pg_get_expr(```pg_node_tree , relation_oid , pretty_bool)	text	反编译一个表达式 的任何 Var 都引用
`pg_get_functiondef(```func_oid)	text	获取一个函数的定
`pg_get_function_arguments(```func_oid)	text	获取函数定义的参
`pg_get_function_identity_arguments(```func_oid)	text	获取参数列表来确 值)
`pg_get_function_result(```func_oid)	text	获取函数的 RETURN
`pg_get_indexdef(```index_oid)	text	获取索引的 CREATE
`pg_get_indexdef(```index_oid , column_no , pretty_bool)	text	获取索引的 CREATE 果 column_no 不为 字段的定义。
pg_get_keywords()	setof record	获取SQL关键字和
`pg_get_ruledef(```rule_oid)	text	为规则获取 CREATE
`pg_get_ruledef(```rule_oid , pretty_bool)	text	为规则获取 CREATE
`pg_get_serial_sequence(```table_name , column_name)	text	获取一 个 serial , smal 段使用的序列名
pg_get_triggerdef (trigger_oid)	text	为触发器获 取 CREATE [CONSTR
pg_get_triggerdef (trigger_oid , pretty_bool)	text	为触发器获 取 CREATE [CONSTR
`pg_get_userbyid(```role_oid)	name	获取给定 OID 的角

`pg_get_viewdef(```view_name)	text	为视图或物化视图命令(已废弃)
`pg_get_viewdef(```view_name , pretty_bool)	text	为视图或物化视图命令(已废弃)
`pg_get_viewdef(```view_oid)	text	为视图或物化视图命令
`pg_get_viewdef(```view_oid , pretty_bool)	text	为视图或物化视图命令
`pg_get_viewdef(```view_oid , wrap_column_int)	text	为视图或物化视图命令；行字段被换到含的
`pg_options_to_table(```reloptions)	setof record	获取存储选项名称
`pg_tablespace_databases(```tablespace_oid)	setof oid	获取在指定的表空集合
`pg_tablespace_location(```tablespace_oid)	text	获取表空间所在的
`pg_typeof(```any)	regtype	获取任何值的数据
`collation for (```any)	text	获取参数的排序

`format_type` 通过某个数据类型的类型 OID 以及可能的类型修饰词返回其 SQL 名称。 如果不知道具体的修饰词，那么在类型修饰词的位置传入 NULL 。

`pg_get_keywords` 返回一组记录描述服务器识别的 SQL 关键字。 `word` 列包含关键字。 `catcode` 列包含一个分类代码： `U` 通用的， `C` 列名， `T` 类型或函数名，或 `R` 保留。 `catdesc` 列包含了一个可能本地化描述分类的字符串。

`pg_get_constraintdef` ， `pg_get_indexdef` ， `pg_get_ruledef` ，和 `pg_get_triggerdef` 分别从一个约束、索引、规则或触发器上重新构造创建它们的命令(反编译的重新构造， 而不是该命令的原文)。 `pg_get_expr` 反编译一个独立表达式的内部形式， 比如一个字段的缺省值。在检查系统表的内容的时候很有用。如果表达式可能包含Var， 那么指定他们参考的关联的OID为第二个参数；如果没有Var，0就足够了。 `pg_get_viewdef` 重新构造出定义视图的 `SELECT` 查询。 这些函数大多数都有两个变种，其中一个是"适合打印"的结果。这种格式更容易读， 但是缺省的格式更有可能被将来的PostgreSQL版本用同样的方法解释； 如果是用于转储，那么尽可能避免使用适合打印的格式。给 pretty-print参数传递 `false` 生成的结果和没有这个参数的变种生成的结果是完全一样。

`pg_get_functiondef` 为函数返回一个完整的 `CREATE OR REPLACE FUNCTION` 语句。 `pg_get_function_arguments` 返回一个函数的参数列表， 这种格式需要在 `CREATE FUNCTION` 中使用。 `pg_get_function_result` 为函数简单的返回适当的 `RETURNS` 子句。 `pg_get_function_identity_arguments` 返回需要的参数列表以标识函数，这种格式需要在 `ALTER FUNCTION` 中使用。这种形式省略了默认值。

`pg_get_serial_sequence` 返回与一个字段相关的序列名字； 如果没有任何序列与给定的字段相关则返回 NULL 。第一个输入参数是可选模式的表名， 第二个参数是列名。因为第一个参数可能是一个模式和表，它不是视为一个双引号的标识符， 意味着默认情况下小写，而第二个参数只是列名称，被视为双引号括起来的，并保留其大小写。 这个名字经过了合适的格式化，可以传递给序列函数(参阅Section 9.16)。 这种相关性可以通过 `ALTER SEQUENCE OWNED BY` 修改或删除。 （其实将这个函数命名为 `pg_get_owned_sequence` 或许更为妥当， 因为它的当前名字反映了它通常用于 `serial` 或 `bigserial` 字段的事实。）

`pg_get_userbyid` 通过角色的 OID 抽取对应的用户名。

`pg_options_to_table` 当通过 `pg_class` . `reloptions` 或 `pg_attribute` . `attoptions` 时返回存储选项名字/值对 （ `option_name` / `option_value` ） 的集合。

`pg_tablespace_databases` 允许检查一个表空间的状况， 它返回在该表空间中保存了对象的数据库 OID 集合。如果这个函数返回数据行， 那么该表空间就是非空的，因此不能删除。要显示该表空间中的特定对象， 你需要把 `pg_tablespace_databases` 返回的数据库标识与 `pg_class` 表连接进行查询。

`pg_describe_object` 返回由目录OID，对象OID和一个（或许0个） 子对象ID指定的数据库对象的描述。这个描述趋向于人类易读，并且可能是经过翻译的，取决于服务器的配置。 这有助于确定一个对象的标识存储在 `pg_depend` 目录中。

`pg_identify_object` 返回一个包含足够信息来唯一的标识由系统OID， 对象OID和一个（或许0个）子对象ID指定的数据库对象的行。这个信息趋向于机器易读，并且从不翻译。 `type` 标识数据库对象的类型； `schema` 是对象属于的模式名，或 `NULL` 表明对象类型不属于模式； `name` 是对象的名字，如果需要就加上双引号， 只有在它可以用作对象的唯一标识符时使用（如果相关，跟随模式名）， 否则为 `NULL` ； `identity` 是完整的对象身份，根据对象类型有精确的格式，并且格式内的每个部分都是模式限定的， 必要时加双引号。

`pg_typeof` 返回传递给它的值的数据类型OID。 这可能有利于故障排除或动态构造SQL查询。函数声明返回OID别名类型的 `regtype` （参阅Section 8.18） ；这意味着它和比较目的的OID相同但显示类型名称。例如：

```
SELECT pg_typeof(33);

 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

表达式 `collation for` 返回传递给它的值的排序。例如

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
------------------
 "de_DE"
(1 row)
```

值可能是引号括起来的并且模式限制的。如果没有为参数表达式排序，那么返回一个null值。如果参数不是排序的类型，那么抛出一个错误。

Table 9-55显示的函数将原来用COMMENT 命令存储的评注抽取出来。如果没有找到，则返回 NULL 。

Table 9-55. 注释信息函数

名字	返回类型	描述
`col_description(```table_oid` , `column_number`)	`text`	获取一个表字段的评注
`obj_description(```object_oid` , `catalog_name`)	`text`	获取一个数据库对象的评注
`obj_description(```object_oid`)	`text`	获取一个数据库对象的评注(已废弃)
`shobj_description(```object_oid` , `catalog_name`)	`text`	获取一个共享数据库对象的评注

`col_description` 返回一个表中字段的评注，它是通过表 OID 和字段号来声明的。`obj_description` 不能用于表字段，因为字段没有自己的 OID 。

带有两个参数的 `obj_description` 返回一个数据库对象的评注， 该对象是通过其 OID 和其所属的系统表名字声明的。比如， `obj_description(123456,'pg_class')` 将返回 OID 为 12345 的表的评注。只带一个参数的 `obj_description` 只要求对象 OID ， 现在已经废弃了，因为我们不再保证 OID 在不同的系统表之间是唯一的， 因此可能会返回错误的评注。

`shobj_description` 和 `obj_description` 差不多， 不同之处仅在于前者用于共享对象。一些系统表是通用于集群中所有数据库的全局表， 因此这些表的评注也是全局存储的。

Table 9-56显示的函数在一个输出形式中提供服务器事务信息。 这些函数的主要用途是为了确定在两个快照之间有哪个事务提交。

Table 9-56. 事务ID和快照

名字	返回类型	描述
`txid_current()`	`bigint`	获取当前事务 ID
`txid_current_snapshot()`	`txid_snapshot`	获取当前快照
`txid_snapshot_xip(```txid_snapshot`)	`setof bigint`	获取在快照中进行中的事务ID
`txid_snapshot_xmax(```txid_snapshot`)	`bigint`	获取快照的 `xmax`
`txid_snapshot_xmin(```txid_snapshot`)	`bigint`	获取快照的 `xmin`
`txid_visible_in_snapshot(```bigint` , `txid_snapshot`)	`boolean`	在快照中事务ID是否可见？(不使用子事务ID)

内部事务 ID 类型(`xid`)是32位，每40亿事务循环。然而这些函数导出一个64位格式， 是使用一个"epoch"计数器扩展，所以在安装过程中不会循环。 这些函数使用的数据类型 `txid_snapshot` ，存储在某时刻事物ID可见性的信息。 其组件描述在Table 9-57。

Table 9-57. 快照组件

名字	描述
`xmin`	最早的事务ID（txid）仍然活动。所有较早事务将是可见提交了， 或者要么死掉回滚了。
`xmax`	首先作为尚未分配的txid。所有大于或等于此的txids作为这时的快照都是尚未开始的，因此不可见。
`xip_list`	在当前快照活动的txids。这个列表只包含在 `xmin` 和 `xmax` 之间的活动txids；有可能活动的txids高于 `xmax` 。 一个 `xmin <= txid < xmax` ， 并且不在这个列表中的txid， 是在快照的这个时间已经完成的，因此要么可见或死掉对应它的提交状态。 这个列表不包含子事务的txids。

`txid_snapshot` 的文本表示为： `_xmin_` : `_xmax_` : `_xip_list_` 。 例如 `10:20:10,14,15` 意思为： `xmin=10, xmax=20, xip_list=10, 14, 15` 。

9.26. 系统管理函数

这节描述的函数用来控制和监视PostgreSQL安装。

9.26.1. 配置设置函数

Table 9-58显示了用于查询和修改运行时配置参数的函数。

Table 9-58. 配置设置函数

名字	返回类型	描述
`current_setting(``setting_name)	`text`	获取当前的设置值
`set_config(``setting_name , new_value , is_local)	`text`	设置参数并返回新值

`current_setting` 用于以查询形式获取 `setting_name` 设置的当前值。它和SQL命令 `SHOW` 是等效的。比如：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

`set_config` 将参数 `setting_name` 设置为 `new_value` 。如果 `is_local` 为 `true` ，那么新值将只应用于当前事务。 如果你希望新值应用于当前会话，那么应该使用 `false` 。 它等效于 SQL 命令 `SET` 。比如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

9.26.2. 服务器信号函数

Table 9-59里的函数向其他服务器进程发送控制信号。 通常这些函数的使用限制为超级用户，除了提到的例外。

Table 9-59. 服务器信号函数

名字	返回类型	描述
`pg_cancel_backend(```pid int)	boolean	取消一个后端的当前查询。您可以对另一个后端执行这个函数， 这个后端有和调用这个函数的用户相同的角色。在所有其他情况下，您必须是超级用户。
pg_reload_conf()	boolean	导致所有服务器进程重新装载它们的配置文件
pg_rotate_logfile()	boolean	滚动服务器的日志文件
`pg_terminate_backend(```pid int)	boolean	终止一个后端。您可以对另一个后端执行这个函数， 这个后端有和调用这个函数的用户相同的角色。在所有其他情况下，您必须是超级用户。

如果成功，这些函数返回 `true` ，否则返回 `false` 。

`pg_cancel_backend` 和 `pg_terminate_backend` 向由 pid 标识的后端进程发送一个信号（分别是SIGINT或SIGTERM）。 一个活动的后端进程的 PID 可以从 `pg_stat_activity` 视图的 `pid` 字段找到，或者在服务器上列出 `postgres` 进程 （在Unix上使用ps或在Windows上使用Task Manager）。 一个活动的后端角色可以从 `pg_stat_activity` 视图的 `usename` 字段找到。

`pg_reload_conf` 给服务器发送一个SIGHUP信号， 导致所有服务器进程重新装载配置文件。

`pg_rotate_logfile` 给日志文件管理器发送信号，告诉它立即切换到一个新的输出文件。 这个函数只有在内建的日志收集器运行的时候才有用，否则根本不存在日志文件管理器子进程。

9.26.3. 备份控制函数

Table 9-60里的函数帮助我们进行在线备份。 这些函数不能在恢复时执行（除了 `pg_is_in_backup` ，`pg_backup_start_time` 和 `pg_xlog_location_diff` ）

Table 9-60. 备份控制函数

名字	返回类型	描述
`pg_create_restore_point(```name text)	text	为执行恢复创建一个命名点 (限制为超级用户)
pg_current_xlog_insert_location()	text	获取当前事务日志的插入位置
pg_current_xlog_location()	text	获取当前事务日志的写入位置
`pg_start_backup(```label text [, fast boolean])	text	准备执行在线备份(限制为超级用户或复制的角色)
pg_stop_backup()	text	完成执行在线备份 (限制为超级用户或复制的角色)
pg_is_in_backup()	bool	如果在线专属备份仍在进行中则为真。
pg_backup_start_time()	timestamp with time zone	获取进行中的在线专属备份的开始时间。
pg_switch_xlog()	text	强制转向一个新的事务日志文件(限制为超级用户)
`pg_xlogfile_name(```location text)	text	将事务日志的位置字符串转换为文件名
`pg_xlogfile_name_offset(```location text)	text , integer	将事务日志的位置字符串转换为文件名并返回在文件中的字节偏移量
`pg_xlog_location_diff(```location text , location text)	numeric	计算两个事务日志位置之间的区别

pg_start_backup 接受一个用户定义的备份标签(通常这是备份转储文件存放地点的名字)。 这个函数向数据库集群的数据目录写入一个备份标签文件(backup_label)，执行一次检查点，然后以文本方式返回备份的事务日志起始位置。用户可以忽略这个返回值，提供它只是为了万一需要的场合。

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

这个函数有第二个可选的类型为 `boolean` 的参数。如果为 `true` ， 那么指定尽可能快的执行 `pg_start_backup` 。这强制一个立即的检查点， 将导致I/O操作有一个尖峰，减缓任何当前执行的查询。

`pg_stop_backup` 删除 `pg_start_backup` 创建的标签文件， 并且在事务日志归档区里创建一个备份历史文件。这个历史文件包含给予 `pg_start_backup` 的标签、备份的事务日志起始与终止位置、备份的起始和终止时间。 返回值是备份的事务日志终止位置(同样也可以忽略)。计算出终止位置后， 当前事务日志的插入点将自动前进到下一个事务日志文件，这样， 结束的事务日志文件可以被立即归档从而完成备份。

`pg_switch_xlog` 移动到下一个事务日志文件，以允许将当前日志文件归档 (假定你使用连续归档)。返回值是刚刚完成的事务日志文件的事务日志结束位置 + 1。 如果自从最后一次事务日志切换以来没有活动的事务日志，那么 `pg_switch_xlog` 什么事也不做，直接返回当前使用的事务日志文件的开始位置。

`pg_create_restore_point` 创建一个可以用作恢复目标的命名的事务日志记录， 并返回相应的事务日志位置。给定的名字可以被recovery_target_name 使用以指定恢复将进行到的点。避免使用相同的名字创建多个恢复点， 因为恢复将在第一个名字匹配恢复目标的位置停止。

`pg_current_xlog_location` 使用与前面那些函数相同的格式显示当前事务日志的写入位置。 类似的， `pg_current_xlog_insert_location` 显示当前事务日志的插入位置。 插入点是事务日志在某个瞬间的"逻辑终点"， 而实际的写入位置则是从服务器内部缓冲区写出时的终点。写入位置是可以从服务器外部检测到的终点， 如果想归档部分完成的事务日志文件，那么这个通常就是你想要的结果。插入点主要用于服务器调试目的。 上述两个函数既是只读操作也不需要超级用户权限。

可以使用 `pg_xlogfile_name_offset` 从前述函数的返回结果中抽取相应的事务日志文件名称和字节偏移量。例如：

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

类似的， `pg_xlogfile_name` 仅仅抽取事务日志文件名称。 如果给定的事务日志位置恰好位于事务日志文件的交界上，这两个函数都返回前一个事务日志文件的名字。 这对于管理事务日志归档来说通常是期望的行为，因为前一个文件是当前最后一个需要归档的文件。

`pg_xlog_location_diff`计算两个事务日志位置之间在字节上的不同。 它可以和`pg_stat_replication`或Table 9-60 里面的一些函数一起使用以获取复制滞后。

有关正确使用这些函数的细节，参阅Section 24.3。

9.26.4. 恢复控制函数

Table 9-61里显示的函数提供了当前备机状态的信息。 这些函数可能在恢复期间或正常运行中执行。

Table 9-61. 恢复信息函数

名字	返回类型	描述
`pg_is_in_recovery()`	`bool`	如果恢复仍然在进行中则返回true。
`pg_last_xlog_receive_location()`	`text`	获取最后一个事务日志接收并通过流媒体复制同步到磁盘的位置。 如果流复制仍在进行，这将单调增加。如果恢复已完成， 那么这个值将保持静止在恢复期间最后接收和同步到磁盘的WAL记录值。 如果不能用流复制，或还没有开始，这个函数返回NULL。
`pg_last_xlog_replay_location()`	`text`	获取最后一个事物日志在恢复时重放的位置。如果恢复仍在进行，这将单调增加。如果恢复已经完成，那么这个值将保持静止在恢复期间最后应用的WAL记录值。当服务已经没有恢复的正常启动时，这个函数返回NULL。
`pg_last_xact_replay_timestamp()`	`timestamp with time zone`	获取最后一个事物在恢复时重放的时间戳。这是为在主节点上生成的事务提交或终止WAL记录的时间。 如果没有事务在恢复时重放，那么这个函数返回NULL。否则，如果恢复仍在进行， 那么这将单调增加。如果恢复已经完成，那么这个值将保持静止在恢复时最后事务应用的值。 当服务已经没有恢复的正常启动时，这个函数返回NULL。

Table 9-62里的函数控制恢复的进程。 这些函数可能只在恢复时被执行。

Table 9-62. 恢复控制函数

名字	返回类型	描述
`pg_is_xlog_replay_paused()`	`bool`	如果恢复暂停则返回true。
`pg_xlog_replay_pause()`	`void`	立即暂停恢复。
`pg_xlog_replay_resume()`	`void`	如果恢复暂停了那么重新启动。

当恢复暂停时，没有进一步的数据库更改。如果是在热备里，所有新的查询将看到相同一致的数据库快照， 并且不会有进一步的查询冲突产生，直到恢复继续。

如果不能使用流复制，那么暂停状态将没有问题的无限的延续。 当流复制正在进行时，将连续接收WAL记录，这将最终填满可用磁盘空间，取决于暂停的持续时间， WAL生成的速度和可用的磁盘空间。

9.26.5. 快照同步函数

PostgreSQL允许数据库会话同步他们的快照。*snapshot* 决定哪个数据对于使用这个快照的事务是可见的。当两个或更多会话需要查看数据库中相同的内容时， 快照同步是必须的。如果两个会话只是单独的启动它们的事务，仍然可能有某些事务在这两个 `START TRANSACTION` 命令执行之间提交，所以一个会话看到了那个事务的影响而另外一个没有看到。

要解决这些问题，PostgreSQL允许一个事务*export* 它正在使用的快照。只要导出事务保持打开，其他事务可以*import* 它的快照，因此来保证他们看到的是与第一个事务看到的完全相同的数据库视图。 但是要注意的是，由任一这些事务做出的任何数据库更改对其他事务保持不可见， 对由未提交的事务做出的更改同样适用。所以事务是与已经存在的数据同步的， 但是对它们自己做的更改正常动作。

快照是由 `pg_export_snapshot` 函数输出的，在Table 9-63 里面显示，并且是由SET TRANSACTION命令输入的。

Table 9-63. 快照同步函数

名字	返回类型	描述
`pg_export_snapshot()`	`text`	保存当前的快照并返回它的标识符

函数 `pg_export_snapshot` 保存当前的快照并返回一个 `text` 字符串标识这个快照。 这个字符串必须传递（在数据库外面）给想要导入快照的客户端。这个快照只在事务结束输出它之前是可以导入的。 如果需要的话，一个事务可以输出多个快照。请注意，这样做只在 `READ COMMITTED` 事务中有用， 因为在 `REPEATABLE READ` 和更高的隔离级别，事务在他们的生存周期中使用相同的快照。 一旦一个事务已经输出了任何的快照，它就不能使用PREPARE TRANSACTION做好准备了。

参阅SET TRANSACTION获取如何使用一个输出的快照的信息。

9.26.6. 数据库对象管理函数

Table 9-64里显示的函数计算数据库对象使用的磁盘空间。

Table 9-64. 数据库对象尺寸函数

名字	返回类型	描述
`pg_column_size(``any`)	`int`	存储一个指定的数值需要的字节数(可能压缩过)
`pg_database_size(``oid`)	`bigint`	指定 OID 代表的数据库使用的磁盘空间
`pg_database_size(``name`)	`bigint`	指定名称的数据库使用的磁盘空间
`pg_indexes_size(``regclass`)	`bigint`	附加到指定表的索引使用的总磁盘空间
`pg_relation_size(``relation regclass`, `fork text`)	`bigint`	指定表或索引的指定分叉树(`'main'`, `'fsm'` 或 `'vm'`)使用的磁盘空间
`pg_relation_size(``relation regclass`)	`bigint`	`pg_relation_size(..., 'main')`的简写
`pg_size_pretty(``bigint`)	`text`	把用64位整数表示的字节计算的尺寸转换成一个人类易读的尺寸
`pg_size_pretty(``numeric`)	`text`	把用数值表示的字节计算的尺寸转换成一个人类易读的尺寸
`pg_table_size(``regclass`)	`bigint`	指定的表使用的磁盘空间，不计索引（但是包含TOAST，自由空间映射和可见性映射）
`pg_tablespace_size(``oid`)	`bigint`	指定 OID 代表的表空间使用的磁盘空间
`pg_tablespace_size(``name`)	`bigint`	指定名字的表空间使用的磁盘空间
`pg_total_relation_size(``regclass`)	`bigint`	指定的表使用的总磁盘空间，包括所有的索引和TOAST数据

`pg_column_size` 显示用于存储某个独立数据值的空间。

`pg_total_relation_size` 接受一个表或压缩表的OID或名字， 并且返回那个表使用的总的在磁盘上的空间，包括所有相关的索引。 这个函数相当于 `pg_table_size` `+` `pg_indexes_size`

`pg_table_size` 接受一个表的OID或名字，并且返回那个表需要的磁盘空间， 不包括索引。（包含TOAST空间，自由空间映射和可见性映射）

`pg_indexes_size` 接受一个表的OID或名字， 并且返回所有附加到这个表上的索引使用的总的磁盘空间。

`pg_database_size` 和 `pg_tablespace_size` 接受一个数据库或表空间的OID或名字，并且返回该对象使用的总的磁盘空间。

`pg_relation_size` 接受一个表、索引、压缩表的 OID 或者名字， 然后返回它们以字节计的磁盘大小。指定 `'main'` 或省略第二个参数返回这个关系的主数据支路的大小。指定 `'fsm'` 返回和这个关系有关的自由空间映射（参阅Section 58.3）的大小。 指定 `'vm'` 返回和这个关系有关的可见性映射 （参阅Section 58.4）的大小。请注意，这个函数只显示一个支路的大小；更多的是想更方便的使用高级函数 `pg_total_relation_size` 或 `pg_table_size` 。

`pg_size_pretty` 用于把其它函数的结果格式化成一种人类易读的格式， 可以根据情况使用KB 、MB 、GB 、TB 。

以上操作在表或索引上的函数接受一个 `regclass` 参数，这个参数简单的是表的OID 或 `pg_class` 系统目录中的索引。你不需要手动的去查看OID， 因为 `regclass` 数据类型的输入转换将为你做这件事。只需要写下包含在单引号中的表名， 这样看起来像是一个字符串常量。为了与普通的SQL名字的处理兼容， 这个字符串将被转换成小写，除非表名用双引号括起。

如果一个不代表活动对象的OID传递给以上一个函数的参数，那么返回NULL。

Table 9-65里显示的函数帮助标识指定的与数据库对象有关的磁盘文件。

Table 9-65. 数据库对象位置函数

名字	返回类型	描述
`pg_relation_filenode(``relation regclass)	`oid`	指定关系的文件节点数
`pg_relation_filepath(``relation regclass)	`text`	指定关系的文件路径名

`pg_relation_filenode` 接受一个表、索引、序列或压缩表的OID或者名字， 并且返回当前分配给它的"filenode"数。文件节点是关系使用的文件名字的基本组件 （参阅Section 58.1获取更多信息）。对大多数表来说， 结果和 `pg_class` . `relfilenode` 相同，但对确定的系统目录来说， `relfilenode` 为0而且这个函数必须用来获取正确的值。 如果传递一个没有存储的关系，比如一个视图，那么这个函数返回NULL。

`pg_relation_filepath` 类似于 `pg_relation_filenode` ， 但是它返回关系的整个文件路径名（相对于数据库集群的数据目录 `PGDATA` ）。

9.26.7. 通用文件访问函数

Table 9-66 里的函数提供了对数据库服务器所在机器上的文件的本地访问接口。 只有那些在数据库集群目录和 `log_directory` 目录里面的文件可以访问。 使用相对路径访问集群目录里面的文件，以及匹配 `log_directory` 配置设置的路径访问日志文件。只有超级用户才能使用这些函数。

Table 9-66. 通用文件访问函数

名字	返回类型	描述
`pg_ls_dir(``dirname text)	setof text	列出目录中的文件
`pg_read_file(``filename text [, offset bigint , length bigint])	text	返回一个文本文件的内容
`pg_read_binary_file(``filename text [, offset bigint , length bigint])	bytea	返回一个文件的内容
`pg_stat_file(``filename text)	record	返回一个文件的信息

pg_ls_dir 返回指定目录里面的除了特殊项" . " 和" .. "之外的所有名字。

pg_read_file 返回一个文本文件的一部分，从 offset 开始， 返回最多 length 字节(如果先达到文件结尾，则小于这个数值)。 如果 offset 是负数，那么它就是相对于文件结尾回退的长度。 如果省略了 offset 和 length ，则返回整个文件。 从文件读取到的字节在服务器编码里被解释为一个字符串； 如果它们在那种编码下是不可用的则抛出一个错误。

pg_read_binary_file 类似于 pg_read_file ， 除了结果是 bytea 值；因此，不执行编码检查。 与 convert_from 函数结合，这个函数可以用来读取用指定编码的一个文件。

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_file 返回一个记录，这个记录包含文件大小， 最后访问的时间戳，最后修改的时间戳，最后文件状态改变的时间戳（只在Unix平台上）， 文件创建的时间戳（只在Windows），和一个 boolean 表明是否为一个路径。 典型的用法包括：

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.8. 咨询锁函数

Table 9-67中的函数用于管理咨询锁(Advisory Lock)。 有关正确使用这些函数的细节，参阅Section 13.3.4。

Table 9-67. 咨询锁函数

名字	返回类型	描述
`pg_advisory_lock(``key bigint)	void	获取排他会话级别咨询锁
`pg_advisory_lock(``key1 int , key2		

int)		
`pg_advisory_lock_shared(```key bigint)	void	获取共享会话级别咨询锁
`pg_advisory_lock_shared(```key1 int , key2 int)	void	获取共享会话级别咨询锁
`pg_advisory_unlock(```key bigint)	boolean	释放一个排他会话级别咨询锁
`pg_advisory_unlock(```key1 int , key2 int)	boolean	释放一个排他会话级别咨询锁
pg_advisory_unlock_all()	void	释放所有当前会话持有的会话级别咨询锁
`pg_advisory_unlock_shared(```key bigint)	boolean	释放一个共享会话级别咨询锁
`pg_advisory_unlock_shared(```key1 int , key2 int)	boolean	释放一个共享会话级别咨询锁
`pg_advisory_xact_lock(```key bigint)	void	获取排他事务级别咨询锁
`pg_advisory_xact_lock(```key1 int , key2 int)	void	获取排他事务级别咨询锁
`pg_advisory_xact_lock_shared(```key bigint)	void	获取共享事务级别咨询锁
`pg_advisory_xact_lock_shared(```key1 int , key2 int)	void	获取共享事务级别咨询锁
`pg_try_advisory_lock(```key bigint)	boolean	尝试获取排他会话级别咨询锁
`pg_try_advisory_lock(```key1 int , key2 int)	boolean	尝试获取排他会话级别咨询锁
`pg_try_advisory_lock_shared(```key bigint)	boolean	尝试获取共享会话级别咨询锁
`pg_try_advisory_lock_shared(```key1 int , key2 int)	boolean	尝试获取共享会话级别咨询锁
`pg_try_advisory_xact_lock(```key bigint)	boolean	尝试获取排他事务级别咨询锁
`pg_try_advisory_xact_lock(```key1 int , key2 int)	boolean	尝试获取排他事务级别咨询锁
`pg_try_advisory_xact_lock_shared(```key bigint)	boolean	尝试获取共享事务级别咨询锁
`pg_try_advisory_xact_lock_shared(```key1 int , key2 int)	boolean	尝试获取共享事务级别咨询锁

`pg_advisory_lock` 锁定一个应用程序定义的资源， 该资源可以用一个 64 位或两个不重叠的 32 位键值标识。如果已经有另外的会话锁定了该资源， 那么该函数将会阻塞到该资源可用为止。这个锁是排它的。多个锁定请求将会被压入栈中，因此， 如果同一个资源被锁定了三次，那么它必须被解锁三次以将资源释放给其它会话使用。

`pg_advisory_lock_shared` 类似于 `pg_advisory_lock` ， 不同之处仅在于共享锁可以和其它请求共享锁的会话共享，但排他锁除外。

`pg_try_advisory_lock` 类似于 `pg_advisory_lock` ， 不同之处在于该函数不会阻塞以等待资源的释放。它要么立即获得锁并返回 `true` ， 要么返回 `false` 表示目前不能锁定。

`pg_try_advisory_lock_shared` 类似于 `pg_try_advisory_lock` ， 不同之处在于该函数尝试获得一个共享锁而不是一个排它锁。

`pg_advisory_unlock` 释放先前取得的排他会话级别咨询锁。 如果释放成功则返回 `true` 。如果指定的锁并未持有， 那么它将返回 `false` 并且服务器会报告一条 SQL 警告信息。

`pg_advisory_unlock_shared` 类似于 `pg_advisory_unlock` ， 不同之处在于该函数释放的是共享会话级别咨询锁。

`pg_advisory_unlock_all` 将会释放当前会话持有的所有会话级别咨询锁， 该函数在会话结束的时候被隐含调用，即使客户端异常地断开连接也是一样。

`pg_advisory_xact_lock` 类似于 `pg_advisory_lock` ， 不同之处在于锁是自动在当前事务的结束释放的，而且不能被显式的释放。

`pg_advisory_xact_lock_shared` 类似于 `pg_advisory_lock_shared` ， 不同之处在于锁是自动在当前事务的结束释放的，而且不能被显式的释放。

`pg_try_advisory_xact_lock` 类似于 `pg_try_advisory_lock` ， 不同之处在于锁，如果得到，是自动在当前事务的结束释放的，而且不能被显式的释放。

`pg_try_advisory_xact_lock_shared` 类似于 `pg_try_advisory_lock_shared` ， 不同之处在于锁，如果得到，是自动在当前事务的结束释放的，而且不能被显式的释放。

9.27. 触发器函数

当前PostgreSQL提供一个内建的触发器函数，`suppress_redundant_updates_trigger`， 其将阻止任何不会实际更改行中的数据发生，相反不管数据是否已经改变始终执行的更新这种不正常的行为。 （这是正常的行为，使得更新运行速度更快，因为不需要检查，并在某些情况下也是有用的。）

理想的情况下，你通常应该避免运行实际上并没有改变记录中的数据的更新。 冗余更新会花费大量不必要的时间，尤其是如果有大量索引要改变， 并且最终将不得不清空死行中的空间。然而，在客户端代码检测这种情况并不总是容易的或甚至可能的， 而写表达式以检测到它们容易产生错误。另一种方法是使用`suppress_redundant_updates_trigger`， 它可以跳过不改变数据的更新。不过你应该小心使用这个命令。触发器为每条记录花费小但有意义的时间， 所以如果更新实际改变会影响大多数记录，那么此触发器的使用将实际上使更新运行得更慢。

`suppress_redundant_updates_trigger`函数可以添加到一个表：

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

在大多数情况下，你可能想要在每行的后面触发这个触发器。记住触发器是以名字的顺序触发的， 你应该在表中你有的任何其他触发器后面选择一个触发器名字。

更多有关创建触发器的信息请参阅CREATE TRIGGER。

9.28. 事件触发函数

当前PostgreSQL提供一个内建的事件触发帮助函数 `pg_event_trigger_dropped_objects` 。

`pg_event_trigger_dropped_objects` 返回一个在 `sql_drop` 事件中调用的命令删除的所有对象的列表。如果在任何其他上下文中调用， `pg_event_trigger_dropped_objects` 将抛出一个错误。 `pg_event_trigger_dropped_objects` 返回下列的字段：

名字	类型	描述
`classid`	`Oid`	对象所在的目录的OID
`objid`	`Oid`	目录中对象的OID
`objsubid`	`int32`	对象的sub-id(例如，字段的属性个数)
`object_type`	`text`	对象的类型
`schema_name`	`text`	如果有，为对象所在模式的名字；否则为 `NULL` 。不用双引号。
`object_name`	`text`	如果模式和名字的组合可以用来唯一的标识对象，那么就是对象的名字；否则为 `NULL` 。 不用双引号，并且名字是从不模式限定的。
`object_identity`	`text`	对象身份的文本表现，模式限定的。每个标识符在身份中出现时要在必要时引用。

`pg_event_trigger_dropped_objects` 函数可以在一个事务触发器中使用：

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE PROCEDURE test_event_trigger_for_drops();
```

关于事务触发器的更多信息请参阅Chapter 37。

Chapter 10. 类型转换

Table of Contents

SQL语句可能(有意无意地)要求在同一表达式里混合不同的数据类型。 PostgreSQL在计算混合类型表达式方面有许多扩展性很强的功能。

在大多数情况下，用户不需要明白类型转换机制的细节。但是， 由PostgreSQL所进行的隐含类型转换会对查询的结果产生影响， 必要时这些影响又可以用明确类型转换进行剪裁利用。

本章介绍PostgreSQL类型转换的传统和机制。 关于特定的类型和函数及操作符的进一步信息， 请参考Chapter 8 和Chapter 9里的相关章节。

10.1. 概述

SQL是强类型语言。也就是说，每个数据都与一个决定其行为和用法的数据类型相关联。PostgreSQL有一个可扩展的数据类型系统， 该系统比其它SQL实现更具通用性和灵活性。因而， PostgreSQL中大多数类型转换是由通用规则来管理的， 而不是由专门的试探法分析的，这种做法允许使用混合类型的表达式， 即便是其中包含用户定义的类型也如此。

PostgreSQL扫描/分析器只将词法元素分解成五个基本种类： 整数、浮点数、字符串、标识符、关键字。大多数非数字类型首先表征为字符串， SQL语言的定义允许将类型名声明为字符串， 这个机制被PostgreSQL用于保证分析器沿着正确的方向运行。 例如，下面查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

有两个文本常量，类型分别为 `text` 和 `point` 。 如果没有为字符串文本声明类型，该文本先被初始化成一个拥有存储空间的 `unknown` 类型，该类型将在后面描述的晚期阶段分析。

在PostgreSQL分析器里， 有四种基本的SQL元素需要独立的类型转换规则：

函数调用

多数PostgreSQL类型系统是建筑在一套丰富的函数上的。 函数调用可以有一个或多个参数。因为PostgreSQL允许函数重载， 所以函数名自身并不唯一地标识将要调用的函数， 分析器必须根据函数提供的参数类型选择正确的函数。

操作符

PostgreSQL允许在表达式上使用前缀或后缀(单目)操作符， 也允许表达式内部使用双目操作符(两个参数)。像函数一样，操作符也可以被重载， 因此操作符的选择也和函数一样取决于参数类型。

值存储

`INSERT` 和 `UPDATE` 语句将表达式结果放入表中。 语句中的表达式类型必须和目标字段的类型一致或者可以转换为一致。

`UNION` , `CASE` 和相关构造

因为联合 `SELECT` 语句中的所有查询结果必须在一列里显示出来， 所以每个 `SELECT` 子句中的元素类型必须相互匹配并转换成一套统一类型。 类似地，一个 `CASE` 构造的结果表达式必须转换成统一的类型， 这样 `CASE` 表达式自身作为整体有一种已知输出类型。 同样的要求也存在于 `ARRAY` 构造以及 `GREATEST` 和 `LEAST` 函数中。

系统表*casts*存储有关哪种数据类型之间存在哪种转换以及如何执行这些转换的信息。 额外的转换可以由用户通过CREATE CAST命令增加。 这个通常和定义一种新的数据类型一起完成。内置的类型转换集已经经过仔细的雕琢了， 因此最好不要去更改它们。

分析器中还提供了一个额外的搜索器，允许提高对有隐含转换的类型组之间的适当的转换行为的决断。 数据类型分成了几个基本类型分类，包括：`boolean`，`numeric`，`string`，`bitstring`，`datetime`，`timespan`，`geometric`，`network`，user-defined(用户定义)。 （参阅列表Table 47-52；但是要注意的是创建自定义的类型分类也是可能的。） 每种类型都有一种或多种首选类型用于解决类型选择的问题。 小心的选择首选类型和可用的隐含转换，就有可能保证有歧义的表达式（那些有多个候选解析方案的） 可以用有效的方式解决。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐含转换决不能有奇怪的或不可预见的输出。
- 如果一个查询不需要隐含的类型转换，分析器或执行器不应该进行更多的额外操作。 这就是说，任何一个类型匹配、格式清晰的查询不应该在分析器里耗费更多的时间， 也不应该向查询中引入任何不必要的隐含类型转换调用。

  另外，如果一个查询通常使用某个函数进行隐含类型转换，而用户定义了一个有正确参数的函数， 解释器应该使用新函数取代原先旧函数的隐含操作。

10.2. 操作符

下面讲解的过程解释了操作符表达式如何确定引用哪个操作符。 请注意这个过程受被调用操作符的优先级影响，因为这将决定哪个子表达式被用来作为操作符的输入。 参阅Section 4.1.6获取更多信息。

操作符类型解析

1. 从系统表 `pg_operator` 中选出要考虑的操作符。 如果使用了一个不带模式修饰的操作符名(常见的状况)， 那么认为该操作符是那些在当前搜索路径中名字和参数个数都匹配的操作符 (参阅Section 5.7.3)。如果给出一个带修饰的操作符名， 那么只考虑指定模式中的操作符。

    i. 如果搜索路径中找到了多个相同参数类型的操作符，那么只考虑最早出现在路径中的那一个。 但是不同参数类型的操作符将被平等看待，而不管它们在路径中的位置如何。

2. 查找精确接受输入参数类型的操作符。如果找到一个(在一组被考虑的操作符中， 可能只存在一个精确匹配的)，则用之。

    i. 如果一个双目操作符调用中的一个参数是 `unknown` 类型， 则在本次检查中假设其与另一个参数类型相同。包括两个 `unknown` 输入的调用或一个一元带有 `unknown` 输入的操作符，将绝不会在此处找到匹配。

3. 寻找最优匹配。

    i. 抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选操作符。 `unknown` 文本在这种情况下可以转换成任何东西。 如果只剩下一个候选项，则用之，否则继续下一步。

    ii. 遍历所有候选操作符，保留那些输入类型匹配最准确的。此时，域被看作和他们的基本类型相同。 如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

    iii. 遍历所有候选操作符，保留那些需要类型转换时接受(属于输入数据类型的类型范畴的)首选类型位置最多的操作符。 如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

    iv. 如果有任何输入参数是 `unknown` 类型，检查剩余的候选操作符对应参数位置的类型范畴。 在每一个能够接受字符串类型范畴的位置使用 `string` 类型(这种对字符串的偏爱是合适的， 因为 unknown 文本确实像字符串)。另外，如果所有剩下的候选操作符都接受相同的类型范畴， 则选择该类型范畴，否则抛出一个错误(因为在没有更多线索的条件下无法作出正确的选择)。 现在抛弃不接受选定的类型范畴的候选操作

符，然后， 如果任意候选操作符在某个给定的参数位置接受一个首选类型， 则抛弃那些在该参数位置接受非首选类型的候选操作符。 如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

v. 如果同时有 `unknown` 和已知类型的参数，并且所有已知类型的参数都是相同的类型， 那么假设 `unknown` 参数也是那种类型，并检查哪个候选操作符在 `unknown` 参数位置接受那个类型。如果只有一个操作符符合，那么使用它。否则，产生一个错误。

下面是一些例子。

Example 10-1. 阶乘操作符类型解析

在系统表中里只有一个阶乘操作符（后缀 `!` ）， 它以 `bigint` 作为参数。 扫描器给下面查询表达式的参数赋予 `integer` 的初始类型：

```
SELECT 40 ! AS "40 factorial";

                   40 factorial
--------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

分析器对参数做类型转换，查询等效于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 10-2. 字符串连接操作符类型分析

一种字符串风格的语法既可以用于字符串也可以用于复杂的扩展类型。 未声明类型的字符串将被所有可能的候选操作符匹配。

有一个未声明的参数的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

本例中分析器寻找两个参数都是 `text` 的操作符。确实这样的操作符， 因此另一个参数就被认为是 `text` 类型。

下面是连接两个未声明类型的值：

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

因为查询中没有声明任何类型，所以本例中对类型没有任何初始提示。因此， 分析器查找所有候选操作符，发现既存在接受字符串类型范畴的操作符也存在接受位串类型范畴的操作符。 因为字符串类型范畴是首选，所以选择字符串类型范畴的首选类型 `text` 作为解析未知类型文本的声明类型。

Example 10-3. 绝对值和取反操作符类型分析

PostgreSQL操作符表里面有几条记录对应于前缀操作符 `@`， 它们都用于为各种数值类型实现绝对值操作。其中之一用于 `float8` 类型， 它是数值类型范畴中的首选类型。因此，在面对 `unknown` 输入的时候， PostgreSQL会使用该类型：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

此处，系统在应用选定的操作符之前隐式的转换unknown类型的文字为 `float8` 类型。 我们可以验证它是 `float8` 而不是其它类型：

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

另一方面，前缀操作符 `~` (按位取反)只为整数数据类型定义， 而不为 `float8` 定义。因此，如果我们用 `~` 做类似的实验将得到：

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

这是因为系统无法决定几个可能的 `~` 操作符中究竟应该使用哪一个。 我们可以用明确地类型转换来帮它：

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Example 10-4. 数组包含操作符类型分析

这里是解决一个操作符带有一个已知和一个未知类型输入的例子：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

PostgreSQL操作符表有几条记录对应于中缀操作符 `<@` ， 但是只有两个可以在左侧接受一个整数数组的操作符是数组包含(`anyarray` `<@` `anyarray`) 和范围包含(`anyelement` `<@` `anyrange`)的。 因为没有多态的伪类型(参阅Section 8.19)是首选的，所以解析器不能解决这个基础上的歧义。 然而，最后一个解析规则告诉我们，假设未知类型的文字是和另外一个输入相同的类型，也就是，整数数组。 现在只有两个操作符中的一个可以匹配，所以选择数组包含。（如果我们选择了范围包含， 我们将得到一个错误，因为字符串没有正确的格式成为范围的文字。）

10.3. 函数

下面讲解的过程解释了如何在一次函数调用中确定所使用的究竟是哪个函数。

函数类型解析

1. 从系统表 `pg_proc` 中选择要考虑的函数。如果使用了一个不带模式修饰的函数名字， 那么认为该函数是那些在当前搜索路径中名字和参数个数都匹配的函数(参阅Section 5.7.3)。 如果给出一个带修饰的函数名，那么只考虑指定模式中的函数。

   i. 如果搜索路径中找到了多个相同参数类型的函数，那么只考虑最早出现在路径中的那一个。 但是不同参数类型的函数将被平等看待，而不管它们在路径中的位置如何。

   ii. 如果使用一个 `VARIADIC` 数组参数声明一个函数，并且调用时不使用关键字 `VARIADIC` ， 那么该函被认为数组参数被一个或更多它的元素类型的实体代替，并且需要去匹配调用。 经过这样的扩展，这个函数可能有和非可变函数相同的有效参数类型。在这种情况下， 使用在搜索路径中出现比较早的函数，或者如果两个函数在相同的模式中，那么首选非可变的一个。

   iii. 考虑使用有默认参数值的函数来匹配任何省略了零或者多个默认表参数位置的调用。 如果多个这样的函数匹配一个调用，那么使用最早出现在搜索路径中的那个。 如果在非默认位置有两个或者更多带有相同模式相同参数类型这样的函数 （他们的默认参数设置可能有不同），系统将不能确定去选择哪个， 并且如果不能找到更好的函数匹配调用，那么将会产生一个"ambiguous function call"错误。

2. 查找精确接受输入参数类型的函数。如果找到一个(在一组被考虑的函数中， 可能只存在一个精确匹配的)，则用之。包含 `unknown` 类型的函数调用绝不会在此处找到匹配。

3. 如果没有找到精确的匹配，则看看函数调用是否需要一个特殊的类型转换。 如果函数调用只有一个参数并且函数名与某些数据类型的内部名称相同，那么就会出现这种情况。 另外，该函数的参数必须是一个未知类型的文本，或者与某个已命名数据类型二进制兼容， 或者是一个可以通过请求那种类型的I/O函数转换为已命名数据类型。（也就是， 要么可以转换成标准字符串类型，要么可以从标准字符串类型转换而来。）如果符合这些条件， 那么该函数调用被认为是一种 `CAST` 声明。 [1]

4. 寻找最优匹配。

   i. 抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选函数。 `unknown` 文本在这种情况下可以转换成任何东西。如果只剩下一个候选项，则用之，否则继续下一步。

ii. 遍历所有候选函数，保留那些输入类型匹配最准确的。此时， 域被看作和他们的基本类型相同。如果没有一个函数能准确匹配，则保留所有候选。 如果只剩下一个候选项，则用之，否则继续下一步。

iii. 遍历所有候选函数，保留那些需要类型转换时接受(属于输入数据类型的类型范畴的)首选类型位置最多的函数。如果没有接受首选类型的函数，则保留所有候选。 如果只剩下一个候选项，则用之，否则继续下一步。

iv. 如果有任何输入参数是 `unknown` 类型，检查剩余的候选函数对应参数位置的类型范畴。 在每一个能够接受字符串类型范畴的位置使用 `string` 类型(这种对字符串的偏爱是合适的， 因为 unknown 文本确实像字符串)。另外，如果所有剩下的候选函数都接受相同的类型范畴， 则选择该类型范畴，否则抛出一个错误(因为在没有更多线索的条件下无法作出正确的选择)。 现在抛弃不接受选定的类型范畴的候选函数，然后，如果任意候选函数在那个范畴接受一个首选类型， 则抛弃那些在该参数位置接受非首选类型的候选函数。如果没有一个候选符合这些测试则保留所有候选。 如果只有一个候选函数符合，则使用它；否则，继续下一步。

v. 如果同时有 `unknown` 和已知类型的参数，并且所有已知类型的参数有相同的类型，假设 `unknown` 参数也是这种类型，检查哪个候选函数可以在 `unknown` 参数位置接受这种类型。如果正好一个候选符合，那么使用它。否则，产生一个错误。

请注意，"最佳匹配"规则对操作符和对函数的类型分析都是一样的。下面是一些例子。

Example 10-5. 圆整函数参数类型解析

只有一个 `round` 函数有两个参数(第一个是 `numeric` ， 第二个是 `integer`)。所以下面的查询自动把第一个类型为 `integer` 的参数转换成 `numeric` 类型：

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

实际上它被分析器转换成：

```
SELECT round(CAST (4 AS numeric), 4);
```

因为带小数点的数值常量初始时被赋予 `numeric` 类型， 因此下面的查询将不需要类型转换，并且可能会略微高效一些：

```
SELECT round(4.0, 4);
```

Example 10-6. 子字符串函数类型解析

有好几个 `substr` 函数，其中一个接受 `text` 和 `integer` 类型。如果用一个未声明类型的字符串常量调用它， 系统将选择接受 `string` 类型范畴的首选类型 (也就是 `text` 类型)的候选函数。

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

如果该字符串声明为 `varchar` 类型，就像从表中取出来的数据一样， 分析器将试着将其转换成 `text` 类型：

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

被分析器转换后实际上变成：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note:** 分析器从 `pg_cast` 表中了解到 `text` 和 `varchar` 是二进制兼容的，意思是说一个可以传递给接受另一个的函数而不需要做任何物理转换。 因此，在这种情况下， 实际上没有做任何类型转换。

而且，如果以 `integer` 为参数调用函数，分析器将试图将其转换成 `text` 类型：

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

这样是不行的，因为 `integer` 不能隐式的转换为 `text` 。 需要一个明确的转换才行：

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

Notes

[1] 这一步骤的原因是为了支持函数风格的转换声明，防止没有实际转换函数的情况。 如果有一个转换函数，它是按照惯例以它的输出类型命名的，这样就不需要一个特例。 参阅CREATE CAST获取额外的说明。

10.4. 值存储

要插入表中的数值也根据下面的步骤转换成目标列的数据类型。

值存储数据类型解析

1. 查找与目标字段准确的匹配。
2. 试着将表达式直接转换成目标类型。如果已知这两种类型之间存在一个已注册的转换函数，那么直接调用该转换函数即可。如果表达式是一个未知类型文本，该文本字符串的内容将交给目标类型的输入转换过程。
3. 检查一下看看目标类型是否有长度转换。长度转换是一个从某类型到自身的转换。如果在 `pg_cast` 表里面找到一个，那么在存储到目标字段之前先在表达式上应用。这样的转换函数总是接受一个额外的类型为 `integer` 的参数，它接收目标字段的 `atttypmod` 值(实际上是其声明长度，`atttypmod` 的解释随不同的数据类型而不同)，并且它可能接受一个 `boolean` 类型的第三个参数，表示转换是显式的还是隐式的。转换函数负责施加那些长度相关的语义，比如长度检查或者截断。

Example 10-7. `character` 存储类型转换

对一个目标列定义为 `character(20)` 的语句，下面的语句显示存储值的长度正确：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

这里真正发生的事情是两个 unknown 文本缺省解析成 `text`，这样就允许 `||` 操作符解析成 `text` 连接。然后操作符的 `text` 结果转换成 `bpchar` ("空白填充的字符型"，`character` 类型内部名称)以匹配目标字段类型。不过，从 `text` 到 `bpchar` 的转换是二进制兼容的，这样的转换是隐含的并且实际上不做任何函数调用。最后，在系统表里找到长度转换函数 `bpchar(bpchar, integer, boolean)` 并且应用于该操作符的结果和存储的字段长。这个类型相关的函数执行所需的长度检查和额外的空白填充。

10.5. `UNION` , `CASE` 和相关构造

SQL `UNION` 构造必须把那些可能不太相似的类型匹配起来成为一个结果集。 解析算法分别应用于联合查询的每个输出字段。 `INTERSECT` 和 `EXCEPT` 构造对不相同的类型使用和 `UNION` 相同的算法进行解析。 `CASE` , `ARRAY` , `VALUES` , `GREATEST` , 和 `LEAST` 构造也使用同样的算法匹配它的部件表达式并且选择一个结果数据类型。

`UNION` , `CASE` 和相关构造的类型解析

1. 如果所有输入都是相同的类型，并且不是 `unknown` 类型，那么解析成这种类型。 否则，用它们潜在的基本类型替换列表中的域类型。
2. 如果所有输入都是 `unknown` 类型则解析成 `text` 类型 (字符串类型范畴的首选类型)。否则，忽略 `unknown` 输入。
3. 如果非 unknown 输入不属于同一个类型范畴，失败。
4. 如果有，则选取第一个属于该范畴中首选类型的非 unknown 输入类型。
5. 否则，选择最后一个允许所有前面的非unknown输入隐式转换为它的非unknown输入类型。 （总是有这么一种类型，因为至少列表上的第一种类型必须适合这种情况。）
6. 把所有输入转换为所选的类型。如果从给定的输入到所选的类型没有一个转换则失败。

下面是一些例子。

Example 10-8. Union中的待定类型解析

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

这里，unknown 类型文本 `'b'` 将被解析成 `text` 类型。

Example 10-9. 简单Union中的类型解析

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

文本 `1.2` 的类型为 `numeric` ，而且 `integer` 类型的 `1` 可以隐含地转换为 `numeric` ，因此使用这个类型。

Example 10-10. 转置Union中的类型解析

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

这里，因为类型 `real` 不能被隐含转换成 `integer` ，但是 `integer` 可以隐含转换成 `real` ，那么联合的结果类型将是 `real` 。

Chapter 11. 索引

Table of Contents

索引是提高数据库性能的常用方法。 索引可以令数据库服务器以比没有索引快得多的速度查找和检索特定的行。 不过索引也在总体上增加了数据库系统的负荷，因此我们应该恰当地使用它们。

11.1. 介绍

假设有像下面这样一个表：

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

发出大量下面这样的语句进行查询：

```
SELECT content FROM test1 WHERE id = _constant_;
```

通常，数据库系统不得不一行一行地扫描整个 `test1` 表以寻找所有匹配的记录。如果在 `test1` 里面有许多行， 但是只返回少数几行(可能是零行或一行)，那么上面这个方法可就很差劲了。 如果我们让数据库系统在 `id` 列上维护一个索引用于定位匹配的行。 这样，数据库系统只需要在搜索树中走少数的几层就可以找到匹配行。

在大多数非小说的书籍里面都使用了类似这样的方法：在书的背后收集着读者会经常查找的术语和概念的索引， 并按照字母顺序排列。有兴趣的读者可以快速地扫描该索引并且切换到合适的页， 因此不用阅读整本书就能查找到感兴趣的位置。作者的任务之一就是预计哪些项是读者最需要查找的东西， 与之类似，预计哪些索引可以带来便利也是数据库程序员的任务。

下面的命令可以用于在 `id` 列上创建前面讨论过的索引：

```
CREATE INDEX test1_id_index ON test1 (id);
```

索引名字 `test1_id_index` 可以自由选择， 但是应该选那些稍后可以让你回忆起索引含义的名字。

要删除一个索引，使用 `DROP INDEX` 命令。 你可以在任何时候向表里增加索引或者从表中删除索引。

一旦你创建了索引，那么就不需要更多干涉了：当表有修改时系统会更新索引， 并且当系统认为用索引比顺序的表扫描快的时候它就会使用索引。 不过你可能必须经常性地运行 `ANALYZE` 命令以更新统计信息， 好让查询规划器能够做出训练有素的判断。参见Chapter 14 获取关于如何获知是否使用了索引的信息，以及在什么时候、什么原因下规划器会决定 不使用索引。

索引对带搜索条件的 `UPDATE` 和 `DELETE` 命令也有好处。 索引更可以用于表连接查询。因此，如果你定义了索引的列是连接条件的一部分， 那么它也可以显著提高连接的查询速度。

在一个巨大的表上创建索引可能会消耗大量的时间。缺省时，PostgreSQL 允许在创建索引的同时读取表(`SELECT` 语句)，但是写入表(`INSERT`，`UPDATE`，`DELETE`)的动作将被阻塞到索引创建完毕。 在生产环境下这种阻塞通常是不可接受的，因此也允许在创建索引的同时写入表，但是有一些警告需要注意，更多信息参见并发建立索引。

创建索引之后，它必须和表保持同步。这些操作增加了数据操作的负荷。 因此我们应该把那些非关键或者根本用不上的索引删除掉。

11.2. 索引类型

PostgreSQL提供了好几种索引类型：B-tree, Hash, GiST, SP-GiST和GIN 。 每种索引类型都比较适合某些特定的查询类型，因为它们用了不同的算法。缺省时， `CREATE INDEX` 命令将创建 B-tree 索引，它适合大多数情况。

B-tree 适合处理那些能够按顺序存储的数据之上的等于和范围查询。 特别是在一个建立了索引的字段涉及到使用

<
<=
=
>=
>

操作符之一进行比较的时候，PostgreSQL 的查询规划器都会考虑使用 B-tree 索引。等效于这些操作符组合的构造， 比如 `BETWEEN` 和 `IN` ，也可以用搜索 B-tree 索引实现。 同样，索引列中的 `IS NULL` 或 `IS NOT NULL` 条件可以和B-tree索引一起使用。

仅当模式是一个常量，并且锚定在字符串开头的时候， 优化器才会把 B-tree 索引用于模式匹配操作符 `LIKE` 和 `~` ， 比如： `col LIKE 'foo%'` 或 `col ~ '^foo'` ， 但是 `col LIKE '%bar'` 就不行。同时，如果你的数据库未使用 C 区域设置， 那么你需要用一个特殊的操作符类创建索引来支持模式匹配查询上的索引。 参阅Section 11.9。还有可能将 B-tree 索引用于 `ILIKE` 和 `~*` ，但是仅当模式以非字母字符(不受大小写影响的字符)开头才可以。

B-tree索引也可以用来按照排序顺序检索数据。这并不总是比一个简单的扫描和排序快， 但通常是有帮助的。

Hash 索引只能处理简单的等于比较。当一个索引了的列涉及到使用 `=` 操作符进行比较的时候， 查询规划器会考虑使用 Hash 索引。下面的命令用于创建 Hash 索引：

```
CREATE INDEX _name_ ON _table_ USING hash (_column_);
```

Caution

Hash 索引操作目前没有记录 WAL 日志，因此如果数据库崩溃有未写入的改变， 我们可能需要用 `REINDEX` 重建 Hash 索引。另外， 对hash索引的改变在初始的基础备份后不是基于流复制或者基于文件复制的， 所以对于随后使用它们的查询会给出错误的回复。因为这些原因， 我们并不鼓励使用 Hash 索引。

GiST 索引不是单独一种索引类型，而是一种架构，可以在这种架构上实现很多不同的索引策略。 因此，可以使用 GiST 索引的特定操作符类型高度依赖于索引策略(操作符类)。 作为示例，PostgreSQL的标准发布中包含用于二维几何数据类型的 GiST 操作符类，它支持

| `<<` || `&<` || `&>` || `>>` || `<<|` || `&<|` || `|&>` || `|>>` || `@>` || `<@` || `~=` || `&&` |

操作符的索引查询。这些操作符的含义参见Section 9.11。 许多其它 GiST 操作符类可用于 `contrib` 中，或者是单独的项目， 更多信息参见Chapter 55。

GiST索引也可最优化"nearest-neighbor"检索，例如

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

找出距离给出目标点最近的十个地点。能这样做也是依赖于使用特定的操作符类。

SP-GiST索引类似于GiST索引，提供一个支持不同类型检索的架构。 SP-GiST允许广泛不同的非平衡基于磁盘的数据结构的实施，例如四叉树，k-d树和根树(尝试)。 作为示例，PostgreSQL的标准发布中包含用于二维点的SP-GiST操作符类， 它支持

| `<<` || `>>` || `~=` || `<@` || `<^` || `>^` |

操作符的索引查询。（这些操作符的含义参见Section 9.11。） 更多信息参见Chapter 56。

GIN 索引是反转索引，它可以处理包含多个键的值(比如数组)。与 GiST和SP-GiST 类似，GIN 支持用户定义的索引策略，可以使用 GIN 索引的特定操作符类型根据索引策略的不同而不同。 作为示例，PostgreSQL的标准发布中包含用于一维数组的 GIN 操作符类，它支持

| `<@` || `@>` || `=` || `&&` |

操作符的索引查询。这些操作符的含义参见Section 9.18。 许多其它 GIN 操作符类可用于 `contrib` 集合或作为单独的项目。 更多信息参见Chapter 57。

11.3. 多字段索引

一个索引可以定义在表中多个字段上。比如下面这样的表(把 `/dev` 目录保存在一个数据库里)：

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

并且你经常发出下面这样的查询：

```
SELECT name FROM test2 WHERE major = _constant_ AND minor = _constant_;
```

那么在字段 `major` 和 `minor` 上联合定义一个索引是比较合适的做法，也就是：

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

目前，只有 B-tree , GiST和 GIN 支持多字段索引。缺省最多可以声明 32 个字段 (这个限制可以在编译PostgreSQL时改变， 见 `pg_config_manual.h` 文件)。

一个多字段的 B-tree 索引可以用在包含索引字段子集的查询条件里，不过， 如果在前导字段(最左边)上有约束条件，那么效率最高。准确的规则是前导字段上的等于约束， 加上第一个没有等于约束的非等于约束字段，将用于限制所扫描的索引范围。 将检查这两个字段右边字段上的索引以减少对表的访问，但是并不减少需要扫描的索引。比如， 假如我们有一个在 `(a, b, c)` 上的索引，查询条件是 `WHERE a = 5 AND b >= 42 AND c < 77` ， 那么索引就需要先扫描所有 `a` = 5 且 `b` = 42 ， 直到所有 `a` = 5 的记录扫描完毕。那些 `c` >= 77的索引条目将被忽略，但是他们仍然会被扫描。 这个索引原则上仍然会被用于那些在 `b` 和/或 `c` 上有约束， 但是在 `a` 上没有约束的查询，但是就必须扫描整个索引了。因此，在大多数这种情况下， 优化器会选择顺序扫描表，而不使用索引。

一个多字段的 GiST 索引可以用于那些查询条件包含索引字段子集的查询中。 附加字段上的条件会限制索引返回的条目， 但是第一个字段上的条件是决定需要扫描多少索引内容的最重要的字段。 如果在第一个字段上只有很少的一些唯一的数值，那么 GiST 就相对来说不那么高效了， 即使在附加字段上有许多独立的数值也如此。

一个多字段的 GIN 索引可以用于那些查询条件包含索引字段子集的查询中。 不像B-tree 或 GiST，除了查询条件使用的索引字段外，索引的搜索效率是相同的。

当然，每个字段都必须和适合该索引类型的操作符一起使用；包含其它操作符的子句将不会被考虑。

使用多字段索引应该谨慎。在大多数情况下，在单字段上的索引就足够了，并且还节约时间和空间。 除非表的使用模式非常固定，否则超过三个字段的索引几乎没什么用处。 又见 Section 11.5获取有关不同索引设置的优缺点的讨论。

11.4. 索引和 `ORDER BY`

除了只是返回查询到的行，索引可以以一个特定的顺序传送它们。这样就允许查询的 `ORDER BY` 说明可以不用一个单独的排序步骤。当前PostgreSQL支持的索引类型， 只有 B-tree 可以产生排序的输出—其他的索引类型返回的行是非指定的、依赖于实现的顺序。

规划器将考虑满足 `ORDER BY` 声明，通过扫描匹配声明的可用的索引， 或者通过扫描物理顺序的表和做一个明确的排序。对于一个需要扫描表的一大部分的查询， 明确的排序可能要比使用索引快的多，因为它使用顺序存取模式所以需要较少的磁盘I/O。 当只需要获取几行时，索引是更有效的。一个重要的特殊情况是 `ORDER BY` 和 `LIMIT` `_n_` 一起使用:一个明确的排序将处理所有的数据以识别前 `_n_` 行，但是如果有一个索引匹配 `ORDER BY` ，那么前 `_n_` 行可以直接找出，而不用扫描剩下的部分。

默认的，B-tree索引以递增、空值最后的顺序存储记录。这意味着在字段 `x` 上向前扫描索引产生的输出满足 `ORDER BY x` （或者 `ORDER BY x ASC NULLS LAST` ）。 索引扫描也可以向后扫描，产生的输出满足 `ORDER BY x DESC` （或者 `ORDER BY x DESC NULLS FIRST` ，因为 `NULLS FIRST` 默认是 `ORDER BY DESC` ）。

创建索引时，可以通过包含选项 `ASC` , `DESC` , `NULLS FIRST` , 和/或 `NULLS LAST` 调整B-tree索引的顺序；例如：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

以递增顺序、空值在前的顺序存储的索引可以满足 `ORDER BY x ASC NULLS FIRST` 或 `ORDER BY x DESC NULLS LAST` ，取决于扫描的方向。

你可能想知道为什么麻烦的提供所有的四个选项，当向后扫描时两个选项可以包含 `ORDER BY` 的所有变体。在单字段索引中，这些选项确实冗余，但是在多字段索引中，它们就是有用的了。 考虑一个在 `(x, y)` 上的两字段索引：当我们向前扫描时，可以满足 `ORDER BY x, y` ， 或者当我们向后扫描时，可以满足 `ORDER BY x DESC, y DESC` 。但是可能应用经常的需要使用 `ORDER BY x ASC, y DESC` 。在普通的索引上无法得到这种顺序，但是如果索引定义为 `(x ASC, y DESC)` 或 `(x DESC, y ASC)` 就是可能的了。

明显的，没有默认排序顺序的索引是比较专业的特征，但是有时它们对特定的查询可以产生极大的加速。 是否值得维持这样的索引取决于你使用需要特殊排序顺序的查询的频率。

11.5. 组合多个索引

一个单独的索引扫描只能用于这样的条件子句：使用被索引字段和索引操作符类中操作符，并这些条件以 `AND` 连接。假设在 `(a, b)` 上有一个索引， 那么类似 `WHERE a = 5 AND b = 6` 的条件可以使用索引，但是像 `WHERE a = 5 OR b = 6` 的条件就不能直接使用索引。

幸运的，PostgreSQL能够组合多个索引(包括同一索引的多次使用) 来处理单个索引扫描不能实现的情况。系统可以在多个索引扫描之间组成 `AND` 和 `OR` 条件。比如，一个类似 `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` 这样的查询可以分解成四个在 `x` 上的独立扫描，每个扫描使用一个条件， 最后将这些扫描的结果 OR 在一起，生成最终结果。另外一个例子是，如果我们在 `x` 和 `y` 上有独立的索引，一个类似 `WHERE x = 5 AND y = 6` 这样的查询可以分解为几个使用独立索引的子句，然后把这几个结果 AND 在一起，生成最终结果。

为了组合多个索引，系统扫描每个需要的索引，然后在内存里组织一个位图， 它给出索引扫描报告中符合索引条件的表数据行位置。然后，根据查询的需要， 把这个位图使用 AND 和 OR 合并在一起。最后，访问实际的表检索并返回数据行。 表的数据行是按照物理顺序进行访问的，因为那就是位图的布局；这就意味着任何原来的索引排序都将消失，而如果查询有一个 `ORDER BY` 子句， 那么还会有一个额外的排序步骤。因为这个原因，以及每个额外的索引扫描都增加了额外的时间， 规划器有时候会选择使用简单的索引扫描，即使有多个索引可用也如此。

在大多数最简单的应用里，可能有多种索引组合都是有用的，数据库开发人员必须在使用哪个索引之间作出平衡。 有时候多字段索引是最好的，有时候创建一个独立索引并依靠索引组合是最好的。比如， 假如你的查询有时候只涉及字段 `x` ，有时候只涉及字段 `y` ，有时候两个字段都涉及， 那么你可能会选择在 `x` 和 `y` 上创建两个独立的索引， 然后依靠索引组合来处理同时使用两个字段的查询。你也可以在 `(x, y)` 上创建一个多字段索引， 它在同时使用两个字段的查询通常比索引组合更高效，但是，正如我们在Section 11.3 里面讨论的，它对那些只包含 `y` 的查询几乎没有用，因此它不能是唯一一个索引。 一个多字段索引和 `y` 上的独立索引可能会更好。因为对那些只涉及 `x` 的查询， 可以使用多字段索引，但是它会更大，因此也比只在 `x` 上的索引更慢。最后一个选择是创建三个索引， 但是这种方法只有在表的更新远比查询少得多，并且所有三种查询都很普遍的情况下才是合理的。 如果其中一种查询比其它的少很多，那么你可能更愿意仅仅创建两种匹配更常见查询的索引。

11.6. 唯一索引

索引还可以用于强迫字段数值的唯一性，或者是多个字段组合值的唯一性。

```
CREATE UNIQUE INDEX _name_ ON _table_ (_column_ [, ...]);
```

目前，只有 B-tree 索引可以声明为唯一。

如果索引声明为唯一的，那么就不允许出现多个索引值相同的行。NULL 值被认为互不相等。一个多字段唯一索引只在多行数据里所有被索引字段都相同时才拒绝。

如果一个表声明了唯一约束或者主键，那么PostgreSQL 自动在组成主键或唯一约束的字段上创建唯一索引(可能是多字段索引)，以强迫这些约束。

> **Note:** 给表增加唯一约束比较好的办法是`ALTER TABLE ... ADD CONSTRAINT`。 用索引强制唯一约束应该认为是一个实现细节，而不应该直接访问。不过， 我们应该知道没有必要在唯一字段上建立索引，那样做只会重复建立自动创建的索引。

11.7. 表达式上的索引

索引并非一定要是一个底层表的字段，还可以是一个函数或者从一个或多个字段计算出来的标量表达式。 这个特性对于快速访问那些基于计算结果的表非常有用。

比如，做大小写无关比较的常用方法是使用 `lower` 函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

如果我们在 `lower(col1)` 函数的结果上定义索引

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

那么上述查询就可以使用该索引。如果我们把这个索引声明为 `UNIQUE` ， 那么它会禁止创建那种 `col1` 数值只是大小写有别或完全相同的数据行。 因此，在表达式上的索引可以用于强制那些无法定义为简单唯一约束的约束。

另外一个例子是，如果我们经常使用下面这样的查询：

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

那么我们就值得创建下面这样的索引：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

`CREATE INDEX` 命令的语法通常要求在索引表达式周围书写圆括弧， 就像我们在第二个例子里显示的那样。如果表达式只是一个函数调用， 那么可以省略，就像我们在第一个例子里显示的那样。

从维护角度来看，索引表达式相对费劲一些，因为在插入数据行或者更新数据行的时候， 都必须为每一行计算生成的表达式。不过，索引表达式不是在索引查找的时候进行计算的， 因为它们已经存储在索引里了。在上面的两个例子里，系统都把查询看做只是 `WHERE indexedcolumn = 'constant'` ，所以搜索的速度等效于任何其它简单的索引查询。 因此，表达式上的索引在检索速度比插入和更新速度更重要的场合下是有用的。

11.8. 部分索引

部分索引是建立在一个表的子集上的索引； 该子集是由一个条件表达式定义的(叫做部分索引的谓词)。 该索引只包含表中那些满足这个谓词的行。部分索引是一个特殊的特性，但是在某些场合很有用。

部分索引的主要动机是为了避免对普通数值(大量重复的数值)建立索引。 因为在普通数值上的查询就算使用索引也没什么好处，那么还不如从索引中剔除这些大量重复的行。 这样可以减小索引尺寸，提高那些真正使用索引的查询的速度。同时它也能提高更新操作的速度， 因为不是所有情况都需要更新索引。Example 11-1 显示了一个潜在的这方面应用的例子。

Example 11-1. 设置一个部分索引以排除普通数值

假设你在数据库中存储 web 服务器的访问日志。大多数访问是从你的组织内部的 IP 地址范围发起的， 但也有一小部分来自其它地方(比如那些通过拨号进行连接的雇员)。如果你主要搜索来自外部访问的 IP， 那么你就不需要对组织子网的 IP 范围进行索引。

假设表像下面这样：

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

要创建符合例子的索引，使用像下面这样的命令：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

一个可以使用这个索引的典型的查询像这样：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

一个不能使用这个索引的查询是：

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

我们通过观察可以看出，这种类型的部分索引要求普通数值是可以预计的。 所以这种部分索引最好用于没有改变的数据分布。索引可以不定期的重建来适应新数据的分布， 但是这增加了维护工作。

另外一个用途在Example 11-2里显示，它把不感兴趣的数值排除在索引之外。 这个结果有与上面列出的同样的优点，但是它完全拒绝了通过索引访问"不感兴趣"的数值， 即使索引扫描可能对那些数据也有利。显然，为这种情况设置部分索引需要非常仔细并且需要大量试验。

Example 11-2. 设置一个部分索引以排除不感兴趣的数值

如果你有一个表，包含已付款和未付款的定单，而未付款的定单只占总表的一小部分并且是经常使用的部分， 那么你可以通过只在未付款定单上创建一个索引来改善性能。创建索引的命令看起来会像这样：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

可能用到这个索引的查询看起来像：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

不过，该索引也可以用于那些完全不涉及 `order_nr` 查询，比如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这个查询不像在 `amount` 字段上的部分索引那么有效， 因为系统必须扫描整个索引。但是，如果未付款的定单相对较少， 那么用这个部分索引找出未付款的定单将会更快些。

请注意下面这个查询无法使用这个索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

定单 3501 可能是已付款也可能是未付款。

Example 11-2还说明了建了索引的字段和谓词中的字段不必相配。 PostgreSQL支持带任意谓词的部分索引，只要只涉及被索引表的字段就行。 不过，我们要记住的是谓词必须和那些希望从该索引中获益的查询条件相匹配。准确说， 只有在系统能够识别出该查询的 `WHERE` 条件在数学上蕴涵了该索引的谓词时， 这个部分索引才能用于该查询。 PostgreSQL还没有智能到可以完全识别那些形式不同但数学上相等的谓词。 做到这样不仅非常困难，而且在实际使用中也可能非常慢。系统可以识别简单的不相等蕴涵， 比如"x < 1"蕴涵"x < 2"；否则，谓词条件必须准确匹配查询的 `WHERE` 条件，不然系统将无法识别该索引是可用的。匹配发生在查

询规划期间， 而不是运行期间。因此，参数化的查询子句必定不会使用部分索引。例如， 一个预先写好的、带有参数的查询可能指定了"x < ?"， 它不可能对所有可能的参数值都蕴涵"x < 2"。

部分索引的第三种用途是禁止在查询中使用索引。如Example 11-3所示， 这里的概念是在表的子集里创建唯一索引。这样就强制在满足谓词的行中保持唯一性， 而并不约束那些不需要唯一的行。

Example 11-3. 设置一个部分唯一索引

假设我们有一个记录测试输出的表。我们希望确保在每个目标和课题的组合中只有一个 "成功"记录，但是可以有任意数量的"不成功"记录。下面是实现方法：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

如果只有少数成功测试而有很多不成功测试，那么这是一种非常有效的实现方法。

最后，部分索引也可以用于取代系统选择的查询规划。同样，如果数据集的分布是比较特定的形状， 那么会导致系统在不该使用索引的时候使用它。在这种情况下， 我们可以把索引设置为在违反规律的查询中不可用。通常PostgreSQL 对索引的使用会做出合理的选择(比如，它在检索普通数值的时候避免使用它， 因此前面的例子实际上只是节约了索引的尺寸，它并不要求避免索引的使用)， 但是如果出现了错误的规划选择那么请提交一个臭虫报告。

请记住一件事：设置一个部分索引表示你至少和查询规划器知道的一样多， 特别是你知道什么场合下索引是有效的。要形成这些知识要求你经验丰富并且理解 PostgreSQL的索引是如何运作的。在大多数情况下， 部分索引对普通索引的优势并不太明显。

更多有关部分索引的信息可以在 部分索引实例 , *POSTGRES中部分索引:研究计划*, 和 *Generalized Partial Indexes* (cached version) 获得。

11.9. 操作符类和操作符族

定义索引的同时可以为索引的每个字段声明一个操作符类。

```
CREATE INDEX _name_ ON _table_ (_column_ _opclass_ [`_sort options_`] [, ...]);
```

这个操作符类指明该索引用于该字段时要使用的操作符。例如，一个在 `int4` 上的 B-tree 索引将使用 `int4_ops` 类；这个操作符类包括用于 `int4` 的比较函数。实际上，字段类型的缺省操作符通常就足够了。拥有操作符类的主要原因是：对于某些数据类型，可能存在多个有意义的索引行为。例如，我们可能想排序两个复数， 既可能通过绝对值，也可能通过实部。我们可以通过为该数据类型定义两个操作符类， 然后在建立索引时选择合适的那个。操作符类决定了基本的排序方式（这个方式可以通过添加排序选项来修改： `COLLATE` ，`ASC` / `DESC` 和/或 `NULLS FIRST` / `NULLS LAST` ）。

除了缺省的以外，还有一些有内置的操作符类：

- `text_pattern_ops` ,`varchar_pattern_ops` 和 `bpchar_pattern_ops` 操作符类分别支持在 `text` ，`varchar` 和 `char` 类型上的 B-tree 索引。 他们与初始的操作符类的区别是数值是严格地逐个字节比较的，而不是根据区域相关的集合规则进行比较。 这样，如果数据库不使用标准的"C"区域设置，那么这些操作符类适用于那些涉及模式匹配表达式(`LIKE` 或者 POSIX 正则表达式)的查询。举一个例子， 你可以像下面这样对一个 `varchar` 字段进行索引：

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  请注意，如果你希望包含普通 `<` , `<=` ，`>` ,或 `>=` 比较的查询使用索引，那么你还应该创建一个使用缺省操作符类的索引。 这样的查询不能使用 `_xxx_` *pattern_ops* 操作符类。 （不过普通相等比较可以使用这个操作符类。）在同一个字段上创建多个使用不同操作符类的索引是可能的。 如果你确实使用了标准的*"C"*区域设置，那么你就不需要`_*xxx*`_pattern_ops 操作符类，因为使用缺省操作符类的索引可以用于 C 区域里面的模式匹配查询。

下面的查询显示所有已定义的操作符类：

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

一个操作符类实际上只是一个名为操作符族的大构造的子集。在这种情况下， 一些数据类型有相同的行为，这对于定义跨数据类型的操作符通常是有用的，并且允许与索引一起使用。为了这样做，每种类型的操作符类必须分入相同的操作符族内。跨类型的操作符是这个族的成员， 但是与族内的任何一个类都没有关系。

这个查询显示所有定义的操作符族和每个族内的所有操作符：

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. 索引和排序

一个索引只能支持一个索引字段的排序。如果多个排序参与，就需要多个索引。

考虑这些语句：

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

索引自动使用底层字段的排序。所以一个下列格式的查询

```
SELECT * FROM test1c WHERE content > _constant_;
```

可以使用这个索引，因为这个比较会默认使用这个字段的排序。然而， 这个索引不能使涉及到一些其他排序的查询加速。所以如果查询是下列格式，那么，

```
SELECT * FROM test1c WHERE content > _constant_ COLLATE "y";
```

一个额外的索引将会建立，以支持 `"y"` 排序，像这样：

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.11. 检查索引的使用

尽管在PostgreSQL里的索引并不需要维护或调节， 但是检查一下哪些索引在实际查询中被使用了仍然非常重要。 检查索引的使用是通过EXPLAIN命令进行的； 为此目的做的应用在Section 14.1里演示。 我们也可以在一个运行的服务器上收集有关索引使用的统计信息，就像Section 27.2里描述的那样。

归纳一个判断需要设置哪些索引的通用过程是很难的。在前面的章节中已经列出了许多典型的例子。 在大多数情况下我们都需要许多试验。本节的剩余部分就是给出一些这方面的窍门。

- 总是先运行ANALYZE命令收集关于表中数值分布的统计信息。 估计一个查询返回的行数需要这个信息，而规划器需要这个行数以便给每个可能的查询规划赋予真实开销值。 如果缺乏任何真实的统计信息，那么就会假设一些缺省数值，那肯定是不准确的。因此，如果还没有运行 `ANALYZE` 就检查一个应用的索引使用状况，那实际上就是一次失败的检查。 参阅Section 23.1.3和Section 23.1.6获取详细信息。

- 使用真实的数据做实验。用测试数据设置索引将告诉你在测试数据中需要什么索引， 而不是在真实数据中。

  最要命的是用很小的数据集。如果从 100000 行中选 1000 行是使用索引的好时机， 那么从 100 行中选 1 行很难说也需要索引，因为 100 行很可能是装在一个磁盘页里面的，因此没有任何查询规划能比通过顺序访问抓取一个磁盘页面更有效。

  做测试数据的时候也要小心，如果应用还不能在生产环境中使用，那么这也是不可避免的。 那些非常相似的数据、完全随机的数据、或者按照排序顺序插入的数据会令统计信息偏离实际数据的特征。

- 如果索引没有得到使用，那么在测试中强制它的使用也许有些价值。 有一些运行时参数可以关闭各种各样的查询规划(在Section 18.7.1中描述)。 比如，关闭顺序扫描(`enable_seqscan`)和嵌套循环(`enable_nestloop`) 将强迫系统使用不同的规划。如果系统仍然选择顺序扫描或者嵌套循环， 那么在为何索引没有得到使用的问题中可能有更基本的问题，比如， 查询条件和索引不匹配等(前面的章节中介绍了什么样的查询可以使用什么样的索引)。

- 如果强制索引用法确实使用了索引，那么就有两种可能：要么是系统选择是正确的： 使用索引实际上并不合适，要么是查询计划的开销计算并不反映现实情况。 这样你就应该对使用和不使用索引的查询进行计时。 这个时候 `EXPLAIN ANALYZE` 命令就很有用了。

- 如果实际情况说明开销计算是错误的，那么仍然有两种可能。 总开销是从每行的每个规划节点的开销乘以每个规划节点的选择性估计计算出来的。 规划节点的开销估计可以用一些运行时参数进行调节(在Section 18.7.2 中描述)。不准确的选择性估计是因为统计信

息不够充分。 我们可以通过调节统计收集参数(参阅ALTER TABLE)提高选择性估计的精度。

如果你没能通过将开销调整得更准确而实现索引的使用，那么你可能不得不求助于明确地强制索引使用。 并且与PostgreSQL开发人员联系并讨论你的情况。

Chapter 12. 全文检索

Table of Contents

12.1. 介绍

全文搜索（或只是文本搜索）提供满足查询的识别自然语言文档的能力， 并且任意地通过相关性查询进行排序。搜索最常见的类型是找到所有包含给定的查询术语的记录，并且以相似性的查 询顺序返回它们。 `query` 和 `similarity` 的概念是非常灵活的，取决于特定的应用。最简单的 搜索认为 `query` 是一组词，并且 `similarity` 为文档中的查询词出现的频率。

文本搜索操作符已经在数据库中存在多年。PostgreSQL为文本数据类型提供 `~` , `~*` , `LIKE` 和 `ILIKE` 操作符，但它们缺乏许多通过现代信息系统要求的必要属性：

- 没有语言的支持，即使是英语。正则表达式是不充分的，因为他们不能很容易地处理派生词， 比如， `satisfies` 和 `satisfy` 。你可能会丢失包含 `satisfies` 的文档， 虽然你可能会发现他们在寻找 `satisfy` 。使用 `OR` 搜索多个派生形式是可能的，但这很繁琐，而且容易出错（有些词可能会有上千的派生词）。
- 他们没有提供搜索结果的分类（排序），当成千的匹配文档被发现时，这使得它们无效。
- 他们往往比较缓慢，因为没有索引的支持，因此他们必须为每一个搜索处理所有文档。

全文索引允许文档被预处理，并且为后边的快速搜索保存一个索引。预处理包括：

- 解析文档__标记。标识不同类别的记号是非常有用的， 例如，数字，词，复合词，电子邮件地址，这样他们可以用不同的方法来处理。原则上令牌类依赖于具体的应用， 但出于大多数的目的，可以使用一组预定义的类。PostgreSQL使用解析器来 执行这一步。提供了一种标准的解析器，以及为特定的需求创造的自定义分析器。
- 转换标记为__词。词是一个字符串，就像一个标记，但它已经标准化， 这样同一个词的不同形式是一样的。例如，标准化几乎总是包括可折叠的大写字母到小写字母，往往涉及删除后缀（如英语中 的 `s` 或者 `es` ）。这允许搜索找到同一个词的不同形式，没有繁琐的输入所有可能的变种。同时，这一步 通常删除屏蔽词，这是很常见的，他们对于搜索无用。（总之，标记是文档文本的原片段，而词汇被认 为是有用的索引和搜索的词。）PostgreSQL使用词典执行这一步。提供各种标准词典， 以及为特定的需求创造的自定义词典。
- 为优化搜索存储预处理文档。比如，每个文档可以表示为标准化词汇排序数组。 伴随着词汇往往为邻近排序存储位置信息，这是理想的。因此包含查询词的"密集"区域的 文档比分散查询词分配到一个更高的顺序。

字典允许细粒度控制如何使用合适的字典规范化标记。你可以：

- 定义不被索引的屏蔽词。
- 使用Ispell映射同义词到一个词。

- 使用同义词词典将短语映射到一个词。
- 使用Ispell词典将词的不同形式映射到一种范式。
- 使用Snowball词根规则将一个词的不同形式映射到一种范式。

一种数据类型 `tsvector` 用于存储预处理文档，以及类型 `tsquery` 表示处理的查询（Section 8.11）。 为这些数据类型提供很多的函数和操作符（Section 9.13），其中最重要的是匹配运算符 `@@`， 将在Section 12.1.2中介绍。 全文搜索可以使用索引进行加速（Section 12.9）。

12.1.1. 文档是什么？

一个文档是全文搜索系统的搜索单元；例如，杂志上的一篇文章或电子邮件消息。 文本搜索引擎必须能够解析文档，而且可以存储它们父文档词（关键词）的联系性。之后， 这些联系用来搜索包含查询词的文档。

在PostgreSQL中搜索，文档通常是一个数据库表中一行的文本字段，或者这些字段的可能组合（级联）， 可能存储在多个表中或者动态地获得。换句话说，一个文档可以由索引的不同部分构成，它不可能随时随地作 为一个整体存储。比如：

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

> **Note:** 注意:实际上，在这些示例查询中， `coalesce` 使用时应防止一个独立的 `NULL` 属性导致整个文档的 `NULL` 结果。

另外一个可能性是在文档系统中作为简单的文本文档存储。在这种情况下，数据库可以用于存储全 文索引并且执行搜索，同时使用一些唯一标识从文件系统中检索文档。然而，从外部检索文件，数据库 需要拥有超级用户权限或者特殊函数支持，因此比把所有数据保存在PostgreSQL中相比较， 这往往不太方便。同时，保持所有的数据在数据库里面允许轻松访问文档的元数据以帮助索引和显示。

为了文本搜索目的，每个文档必须减少到预处理 `tsvector` 格式。在文档的 `tsvector` 表示形式上完整的执行搜索 和排序—当为了显示给用户来选择文档时，只需要检索原文本。因此我们常说的 `tsvector` 作为文档，当然它仅仅是 完整文档的一种紧凑表示。

12.1.2. 基本文本匹配

PostgreSQL中的全文搜索基于匹配算子 `@@` ， 如果一个 `tsvector` (document)匹配一个 `tsquery` (query)， 则返回 `true` 。不管哪个数据类型先被重写：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

正如上面例子表明，一个 `tsquery` 不仅仅是原文本，更多的是一个 `tsvector` 。一个包含搜索条件的 `tsquery` ， 必须是已经标准化的词，并且可能使用AND, OR, 和 NOT操作符连接多个术语（详情请见Section 8.11）。 函数 `to_tsquery` 和 `plainto_tsquery` 在将用户书写文本转换成一个合适的 `tsquery` 是非常有帮助的。 比如通过标准化文本中的词。类似的， `to_tsvector` 用于解析和标准化文档字符串。因此在实践中文本搜索匹配 可能看起来更像这样：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

观察这个匹配可能不会成功，如果写成这样：

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

由于这儿没有发生词 `rats` 的标准化。一个 `tsvector` 的元素是词，假设已经被标准化，所以 `rats` 不匹配 `rat` 。

`@@` 操作符也支持 `text` 输入，允许一个文本字符串的显示转换为 `tsvector` 或者在简单情况下忽略 `tsquery` 。 可用形式是：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

我们已经看到了前面两种，形式 `text` `@@` `tsquery` 等价于 `to_tsvector(x) @@ y` 。 `text` `@@` `text` 等价于 `to_tsvector(x) @@ plainto_tsquery(y)` 。

12.1.3. 配置

上面是所有简单文本搜索例子。如前所述，全文搜索功能还有能力做更多事情：忽略索引某个词（屏蔽词）， 过程同义词和使用复杂解析，比如：不仅仅基于空白格的解析。这些功能通过文本搜索配置控制。 PostgreSQL来自多语言的预先定义的配置，并且你也可以很容易的创建你自己的配置（psql的 `\dF` 命令显示了 所有可用配置）。

在安装期间选择一个合适的配置，并且在 `postgresql.conf` 中相应的设置default_text_search_config。 如果为了整个集群使用同一个文本搜索配置你可以使用 `postgresql.conf` 中的值。为了在集群中使用不同配置，但是在任何其他一个数据库的同一配置中使用 `ALTER DATABASE ... SET` 。否则，你可以在每个会话中设置 `default_text_search_config` 。

每个依赖于配置的文本搜索函数有一个可选的 `regconfig` 参数，因此可以明确声明使用的配置。 仅当忽略这些参数的时候，才使用 `default_text_search_config` 。

为了更方便的建立自定义文本搜索配置，从简单的数据库对象中建立了配置。 PostgreSQL文本搜索功能提供了四种类型的配置相关的数据库对象：

- 文本搜索解析器打破文档标记，并且分类每个标记（比如，词和数字）。
- 文本搜索词典把标记转换成规范格式并且拒绝屏蔽词。
- 文本搜索模板提供潜在的词典功能（一个词典简单的指定一个模板，并且为模板设置参数）。
- 文本搜索配置选择一个解析器，并且使用一系列词典规范化语法分析器产生的标记。

文本搜索解析器和模板是从低层次的C函数建立的；因此它需要C程序能力开发新产品， 并且需要超级用户权限安装到数据库中（在PostgreSQL发布的 `contrib/` 范围内有附加解析器和模板的实例）。 因为词典和配置仅仅参数化并且连接到一些潜在的解析器和模板上，创建一个新的词典或者配置不需要特定的权限。 创建自定义词典和配置实例出现在本章节的后面。

12.2. 表和索引

上一节中的例子说明使用简单常量字符串的全文匹配。本节显示如何搜索表中的数据，选择使用索引。

12.2.1. 搜索表

在不使用索引的情况下也是可以进行全文检索的,一个简单查询,显示出 `title` 从所有 `body` 字段中包含 `friend` 的每一行：

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

这也将找到相关的词，比如 `friends` 和 `friendly` ，因为所有的这些都降低到相同规范化的词。

以上查询指定 `english` 配置是用来解析和规范化字符串。或者我们可以省略配置参数：

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

这个查询将通过default_text_search_config使用配置设置。

复杂一点的例子:检索出最近的10个文档,在 `title` 或者 `body` 字段中包含 `create` 和 `table` 的：

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为了清楚,我们忽略 `coalesce` 函数调用,这需要找到在两个字段之一中包含 `NULL` 的行。

虽然这些查询在没有索引的情况下工作，大多数应用程序会发现这个方法太慢了，除了偶尔的特定搜索。 文本搜索的实际使用通常需要创建索引。

12.2.2. 创建索引

为了加速文本搜索，我们可以创建GIN索引(Section 12.9)：

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

注意，使用 `to_tsvector` 的2-参数版本。唯一的文本搜索功能指定可用于表达式索引的配置名称（节Section 11.7）。 这是因为索引的内容必须不受default_text_search_config的影响。如果他们受到影响，索引内容可能不一致， 因为不同的条目可能包含不同的文本搜索配置创建的 `tsvector` ，并且没有办法猜出是哪个。正确的转储和恢复 这样的一个索引是不可能的。

因为在上述索引中使用 `to_tsvector` 的2-参数版本，只有一个使用带有相同配置名称的 `to_tsvector` 的2-参数版本的查询参考， 它将使用该索引。也就是说， `WHERE to_tsvector('english', body) @@ 'a & b'` 可以使用索引，但 `WHERE to_tsvector(body) @@ 'a & b'` 不能。 这确保将使用一个索引仅仅伴随着用于创建索引的相同配置。

建立更复杂的表达式索引是可能的，配置名称由另一列指定，例如：

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

在 `pgweb` 表中 `config_name` 是一列。这允许在同一索引中混合配置，当记录的配置被用于每个索引条目。 这将是有用的，例如，如果文档集合中包含不同的语言文件。再次，意味着使用索引的查询必须措辞匹配， 例如， `WHERE to_tsvector(config_name, body) @@ 'a & b'` 。

索引甚至可以连接列：

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

另一个方法是创建一个单独的 `tsvector` 列控制 `to_tsvector` 的输出。这个例子是 `title` 和 `body` 的一个级联， 当其他是 `NULL` 的时候，使用 `coalesce` 确保一个字段仍然会被索引：

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

然后我们为加速搜索创建一个GIN索引：

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

现在我们准备执行一个快速全文搜索：

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

当使用一个单独的列存储 `tsvector` 形式时，有必要创建一个触发器以保持 `tsvector` 列当前任何时候 `title` 或者 `body` 的变化。 节Section 12.4.3解释了如何做。

单独列方法比一个表达式索引的优势是它没有必要明确地声明为充分利用索引查询中的文本搜索配置， 正如上面例子所示，查询依赖于 `default_text_search_config` 。另一个优势是搜索比较快速， 因为它没有必要重新进行 `to_tsvector` 调用来验证索引匹配（当使用GIST索引而不是GIN索引的时候， 这是非常重要的，参见节Section 12.9）。表达式索引方法更容易建立，然而，它需要较少的磁盘空间， 因为 `tsvector` 形式没有明确存储。

12.3. 控制文本搜索

为了执行全文搜索，这必须有个函数创建来自文档的 `tsvector` 和来自用户查询的 `tsquery` 。同时，我们需要以有效的顺序返回结果，因此我们需要一个函数比较关于查询相关性的文档。 可以很好的显示结果也是很重要的。PostgreSQL为所有这些函数提供支持。

12.3.1. 解析文档

PostgreSQL中提供了 `to_tsvector` 函数把文档处理成 `tsvector` 数据类型。

```
to_tsvector([ `_config_` `regconfig`, ] _document_ text) returns tsvector
```

`to_tsvector` 解析文本文档为记号，减少标记到词条，并返回一个 `tsvector` ， 其罗列出词条并连同它们文档中的位置。该文档是根据指定的或默认的文本搜索配置处理的。 这里有一个简单的例子：

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面的例子中我们看到结果 `tsvector` 不包含词 `a` ， `on` 或者 `it` 。 `rats` 变成 `rat` ,并且忽略标点符号-。

该 `to_tsvector` 函数内部调用一个分析器，将文档文本分解成记号并指定每个标记的类型。 为每个标记，参阅词典列表（节Section 12.6），列表因不同的标记类型而不同。 第一本词典识别标记发出一个或多个标准词汇 表示标记。例如， `rats` 变成 `rat` 因为字典认为词 `rats` 是 `rat` 的复数形式。有些词被作为屏蔽词（节Section 12.6.1）， 这样它们就会被忽略，因为它们出现得太过频繁以致于搜索中没有用处。在我们的例子中，它们是 `a` , `on` 和 `it` 。 如果列表中没有词典识别标记，那么它也被忽略。在这个例子中，发生在标点符号处 `-` 因为事实上没有词典分配给它的标记类型（ `空间符号` ）， 意味着空间记号永远不会被索引。语法分析器的选择，词典和索引类型的标记是由选定的文本搜索配置决定（节Section 12.7）。 可以在同一个数据库中有多种不同的配置，与提供各种语言的预定义的配置。在我们的例子中，我们使用缺省配置 `english` 为英语。

函数 `setweight` 可以用于标识一个给定权重的 `tsvector` 的词条，权重是字母 `A` , `B` , `C` 或者 `D` 之一。通常标记来自文档不同部分的词条，比如标题正文。之后，这些信息可以用于搜索结果的排序。

因为 `to_tsvector` (`NULL`)将要返回 `空` ， 当字段可能是空的时候，建议使用 `coalesce` 。 这是体系文档中创建 `tsvector` 推荐的方法：

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

我们使用 `setweight` 标记已完成的 `tsvector` 中的每个词的来源，并且使用 `tsvector` 连接操作符 `||` 合并标签化 `tsvector` 的值。 （节Section 12.4.1详细介绍了这些操作）。

12.3.2. 解析查询

PostgreSQL提供了函数 `to_tsquery` 和 `plainto_tsquery` 将查询转换为 `tsquery` 数据类型。 `to_tsquery` 提供比 `plainto_tsquery` 更多的功能，但对其输入不宽容。

```
to_tsquery([ `_config_` `regconfig`, ] _querytext_ text) returns tsquery
```

`to_tsquery` 从 `_querytext_` 中创建一个 `tsquery` ，它必须由布尔运算符 `&` (AND), `|` (OR) 和 `!` (NOT)分离的单个标记组成。这些运算符可以用圆括弧分组。换句话说， `to_tsquery` 输入必须遵循 `tsquery` 输入的一般规律，如节Section 8.11所描述的。不同的是当基本 `tsquery` 输入以标记表面值的时候， `to_tsquery` 使用指定或默认配置规范每个标记到一个词， 并丢弃所有标记依据配置的屏蔽词。比如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

作为基本 `tsquery` 输入，权重（s）可以附属于每个词来限制它只匹配那些权重(s)的 `tsvector` 词。比如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

`*` 也可以附属于一个词来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

这样的词将匹配以给定字符串开头的 `tsvector` 中的任何词。

`to_tsquery` 也可以接受单引用的短语。当配置包括一个可能触发这类短语的同义词词典库的时候是很有用的。 在下面的例子中，一个词库包含规则 `supernovae stars : sn` ：

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

没有引号， `to_tsquery` 会生成标记语法错误，这个标记不是通过AND 或者OR操作符分离的。

```
plainto_tsquery([ `_config_` `regconfig`, ] _querytext_ text) returns tsquery
```

`plainto_tsquery` 变换未格式化的文本 `_querytext_` 到 `tsquery` 。 分析文本并且归一化为 `to_tsvector` ，然后在存在的词之间插入 `&` (AND)布尔算子。

比如：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

请注意， `plainto_tsquery` 无法识别布尔运算符，权重标签，或在其输入中的前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

在这里，所有的输入标点符号作为空格符号丢弃。

12.3.3. 查询结果关注度

相关度试图衡量哪一个文档是检索中最关注的，所以当有很多匹配时，最相关的一个则最先显示。 PostgreSQL提供了两个预定义的相关函数，其中考虑了词法，距离，和结构信息；也就是， 他们考虑查询词在文档中出现的频率，术语在文档中的紧密程度，以及它们在文档中的部分的重要性。 然而，相关性的概念是模糊的，并且是特定应用程序。不同的应用程序可能需要额外的相关信息， 例如，文档的修改时间。内置的相关函数是唯一的例子。为了以满足您的特定需求，你可以写你自己的相关函数和/或与其他因素相结合的结果。

当前可用的两个相关函数：

`ts_rank([``_weights_` `float4[]` , ] `_vector_` `tsvector` , `_query_` `tsquery` [, `_normalization_` `integer` ]) returns `float4`

基于匹配词汇频率的列向量。

`ts_rank_cd([``_weights_` `float4[]` , ] `_vector_` `tsvector` , `_query_` `tsquery` [, `_normalization_` `integer` ]) returns `float4`

这个函数计算给定文档向量和查询的覆盖密度相关性，正如1999年在 杂志"信息处理与管理"中Clarke，Cormack和Tudhope的"一至三项查询相关性排序"描述的一样。

这些函数需要位置信息的输入。因此它不能在"剥离" `tsvector` 值的情况下运行—它将总是返回零。

对于这些函数，可选的 `_weights_` 参数提供权衡词的情况能力或多或少地取决于它们是如何被标记的。 权重阵列指定顺序权重每类词的频率。

```
{D-weight, C-weight, B-weight, A-weight}
```

如果没有提供 `_weights_` ， 则利用这些缺省值：

```
{0.1, 0.2, 0.4, 1.0}
```

通常的权重是用来标记文档特殊区域的词，如标题或最初的摘要， 所以他们有着比文档主体中的词或多或少的重要性。

由于较长的文档有包含查询词的机会，它合理的考虑文档的大小，例如， 带有搜索词的五个实例的百字文档可能比千字文档有更多的相关性。 相关接受一个整数 `_normalization_` 选项，指定文档长度是否以及如何影响它的排序。 整数选项控制一些行为，所以它是一位掩码：您可以使用 `|` （例如， `2|4` ）指定一个或多个行为。

- 0（缺省）表示跟长度大小没有关系
- 1 表示关注度（rank）除以文档长度的对数+1
- 2表示关注度除以文档的长度
- 4表示关注度除以范围内的平均谐波距离，只能使用 `ts_rank_cd` 实现。
- 8表示关注度除以文档中唯一分词的数量
- 16表示关注度除以唯一分词数量的对数+1
- 32表示关注度除以本身+1

如果指定超过一个的标志位，则在列出顺序中应用转换。

需要特别注意的是，相关函数不使用任何全局信息，所以不可能产生一个所需要的1%或100%的公平归一化。 规范化选项32 (`rank/(rank+1)`)可用于所有规模排序到范围零到一之间，当然，这只是一个表面变化；它不会影响搜索结果的排序。

下面是一个例子，仅仅选择排名前十的匹配：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                       |   rank
-------------------------------------------------+----------
 Neutrinos in the Sun                            |      3.1
 The Sudbury Neutrino Detector                   |      2.4
 A MACHO View of Galactic Dark Matter            |  2.01317
 Hot Gas and Dark Matter                         |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter   |  1.90953
 Rafting for Solar Neutrinos                     |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter       |  1.85774
 Hot Gas and Dark Matter                         |   1.6123
 Ice Fishing for Cosmic Neutrinos                |      1.6
 Weak Lensing Distorts the Universe              | 0.818218
```

这是使用归一化排序的相同例子：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                       |        rank
-------------------------------------------------+-------------------
 Neutrinos in the Sun                            | 0.756097569485493
 The Sudbury Neutrino Detector                   | 0.705882361190954
 A MACHO View of Galactic Dark Matter            | 0.668123210574724
 Hot Gas and Dark Matter                         |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter   | 0.656301290640973
 Rafting for Solar Neutrinos                     | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter       | 0.650072921219637
 Hot Gas and Dark Matter                         | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos                | 0.615384618911517
 Weak Lensing Distorts the Universe              | 0.450010798361481
```

排序花费比较多，因为它需要查找每个匹配文档的 `tsvector` ，这与I/O绑定，因此慢。 不幸的是，它几乎是不可能避免因实际的查询往往导致的大量匹配。

12.3.4. 强调结果

为显示搜索结果，合理显示每个文档的一部分以及查询相关性。通常，搜索引擎显示标记搜索条件的文档片段。 PostgreSQL提供了一个函数 `ts_headline` 实现此功能。

```
ts_headline([ `_config_` `regconfig`, ] _document_ text, _query_ tsquery [, `_options_` `
```

`ts_headline` 接受查询文档，并从突显的查询条件的文档中返回一个摘录。配置用来解析 `_config_` 指定的文档； 如果省略 `_config_` ，则使用 `default_text_search_config` 配置。

如果指定一个 `_options_` 字符串，它必须由一个逗号分隔的一个或多个 `_option_``=``_value_` 对组成。可用选项是：

- `StartSel`，`StopSel`:该字符串分隔文档中出现的查询词，以区别于其他摘录词。 如果它们含有空格或逗号，你必须用双引号字符串。
- `MaxWords`，`MinWords`:这些数字决定最长和最短的标题输出。
- `ShortWord`:这个长度或更短的词在标题的开始和结束被丢弃。三个默认值消除了常见英语文章。
- `HighlightAll`:布尔标志；如果为`真`，整个文档将作为标题，忽略了前面的三个参数。
- `MaxFragments`:要显示的文本摘录或片段的最大数量。默认值零选择非片段标题的生成方法。 一个大于零的值选择基于片段的标题生成。此方法查找文本片段与尽可能多的查询词并在查询词周围延伸这些片段。 作为查询词的结果接近每一片段中间，每边都有词。每个片段至多是`MaxWords`，并且长度为`ShortWord`或更短的词在每一个片段开始和结束被丢弃。 如果不是所有的查询词在文档中找到，则文档中开头的`MinWords`单片段将被显示。
- `FragmentDelimiter`:当一个以上的片段显示时，通过字符串分隔这些片段。

任何未声明的选项接受这些缺省：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

比如:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                      ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

`ts_headline`使用原始文档，而不是一个`tsvector`摘要，因此它很慢，应小心使用。 一个典型的错误是，当只显示10个文档时，为每个匹配文档调用`ts_headline`。SQL子查询可以帮忙， 这是一个例子：

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
      FROM apod, to_tsquery('stars') q
      WHERE ti @@ q
      ORDER BY rank DESC
      LIMIT 10) AS foo;
```

12.4. 附加功能

本节描述了连接文本搜索中有用的附加功能和操作符。

12.4.1. 操作文档

节Section 12.3.1显示了原始文本文档如何转换成 `tsvector` 值。 PostgreSQL也提供用于操作已经在 `tsvector` 形式中的文档的函数和操作符。

`tsvector` || `tsvector`

`tsvector` 连接操作符返回一个连接词的向量，以及作为参数给定的2个向量的位置信息。 在连接期间重新获得位置和权重标签。出现在右边向量位置通过左边向量提到的最大位置相抵消， 因此这个结果几乎等同于2个原始文档字符串连接中执行 `to_tsvector` 的结果。（这个等价是不准确的， 因为任何从左边参数中删除的屏蔽词不会影响结果，然而，如果使用文本连接，它们影响右边参数词的位置）。

使用级联中的向量形式而不是在应用 `to_tsvector` 之前连接文本的一个优势是， 你可以使用不同的配置解析文档的不同部分。同时，由于 `setweight` 函数标记所有相同方式给定向量的词汇， 解析文本是必要的，并且如果你想用不同的权重标记文档不同部分，连接前做 `setweight` 。

`setweight(``_vector_` `tsvector` , `_weight_` `"char"` ) returns `tsvector`

`setweight` 返回一个输入向量的拷贝，其中每一个位置用给定的 `_weight_` , `A` , `B` , `C` 或者 `D` 之一进行标记。 （ `D` 是缺省新向量，因此不显示在输出上。）当向量连接时，保留这些标签， 允许一个文档的不同部分的词通过不同相关函数加权。

注意权重标签适用于位置，不是词汇。如果输入向量已经被剥夺了位置，则 `setweight` 不做任何事情。

`length(``_vector_` `tsvector` ) returns `integer`

返回存储在向量中的词的数量。

`strip(``_vector_` `tsvector` ) returns `tsvector`

返回一个向量，其中列出了给定向量的同一词，但它缺乏任何位置和权重信息。 虽然为相关性排序返回的向量比一个未拆分向量用处少，它通常会小得多。

12.4.2. 处理查询

节Section 12.3.2显示了原始文本查询如何转换成`tsquery`值。 PostgreSQL也提供了函数和操作符用于处理已存在`tsquery`形式中的查询

`tsquery` && `tsquery`

返回两个给定查询的与组合。

`tsquery` || `tsquery`

返回两个给定查询的或组合。

`!!` `tsquery`

返回给定查询的反面（非）。

`numnode(``_query_` `tsquery` ) returns `integer`

返回在一个`tsquery`中节点的数目（词加操作符）。决定`_query_`是否有意义（返回> 0），或只包含屏蔽词（返回0），这个函数是很有用的。例子：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(``_query_` `tsquery` ) returns `text`

返回可用于搜索索引的`tsquery`部分。此函数对检测未索引查询是有帮助的，例如那些只包含屏蔽词或否定术语。比如:

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. 查询重写

函数族`ts_rewrite`搜索一个特定的目标查询事件`tsquery`，和替换每个替代子查询。 实际上这个操作是一个子字符串替换的`tsquery`-特定版本。目标和替换组合可以被认为是一个查询重写规则。 一组这样的重写规则可以是一个强大的搜索帮助。例如，你可以使用同义词扩大搜索（例如，`new york`，`big apple`，`nyc`，`gotham`）或缩小搜索一些热点问题的直接用户。在这些特性和同义词词典之间功能上有一些重叠（节Section 12.6.4）。然而， 你可以在不重建索引情况下即时修改重写规则，而更新词库需要重建索引才能有效。

`ts_rewrite (``_query_` `tsquery` , `_target_` `tsquery` , `_substitute_` `tsquery` ) returns `tsquery`

`ts_rewrite` 的这种形式只适用于一个单一的重写规则：无论出现在 `_query_` 的什么地方， `_target_` 通过 `_substitute_` 替换。比如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

`ts_rewrite (``_query_` `tsquery` , `_select_` `text` ) returns `tsquery`

`ts_rewrite` 的这种形式接受起始 `_查询_` 和SQL `_select_` 命令，这是作为一个文本字符串。 `_select_` 必须产生两列 `tsquery` 类型。 `_select_` 结果的每一行，出现的第一个字段的值（目标） 都被当前的 `_query_` 值中的第二个字段值（替代）。比如：

注意，当多个重写规则适用于这种方式时，应用的顺序非常重要；因此在实践中你将需要源查询为 `ORDER BY` 一些排序关键字。

让我们考虑下现实生活中天文的例子。我们将使用表驱动的重写规则扩大查询 `supernovae` ：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

我们可以通过更新表改变重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
---------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

当有许多的重写规则的时候，重写比较缓慢，因为它检查可能匹配的每一个规则。 为过滤掉明显非候选规则，我们可以使用 `tsquery` 类型的包含操作符。在下面的例子中， 我们只选择那些可能与原始查询匹配的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. 自动更新的触发器

当使用单独的列存储文档的 `tsvector` 形式，当文档内容列变化时， 有必要建立一个触发器更新 `tsvector` 列。两个内置的触发器功能可用于此， 或者你可以自定义触发器。

```
tsvector_update_trigger(_tsvector_column_name_, _config_name_, _text_column_name_ [, ...
tsvector_update_trigger_column(_tsvector_column_name_, _config_column_name_, _text_column
```

这些触发器函数自动计算来自一个或多个文本字段的 `tsvector` 列，在 `CREATE TRIGGER` 命令指定的参数控制下。 使用的例子是：

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

创建触发器，在 `title` 或者 `body` 中的任何改变都会自动反映到 `tsv` 中，而不必担心它的应用。

第一个触发器参数必须是被更新的 `tsvector` 字段名。第二个参数指定要进行转换的文本搜索配置。 为 `tsvector_update_trigger` ，配置的名称仅仅是作为第二个触发器参数。它必须是如上所示的模式匹配， 因此触发器的行为在 `search_path` 中不会改变。
为 `tsvector_update_trigger_column` ， 第二个触发器参数是另一个表列的名称，它的类型必须是 `regconfig` 。这允许每行选择进行配置。 剩余的参数（s）是文本列的名称（键入 `text` ，`varchar` 或者 `char` ）。这些将在给定的顺序中提供文档。 空值将被忽略（但其他列仍将被索引）。

这些内置触发器的限制是它们一致对待所有输入列。为了处理不同列—比如， 为权重不同主体的标题—它有必要编写一个自定义触发器。 这是使用PL/pgSQL 作为触发器语言的一个例子：

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

记住当在触发器内部创造 `tsvector` 值时，明确指定配置的名称是很重要的， 所以，该列的内容将通过改变 `default_text_search_config` 而不会受到影响。 如果不这样做可能会导致诸如重载转储之后搜索结果改变的问题。

12.4.4. 收集文献统计

函数 `ts_stat` 可用于检查你的配置和查找屏蔽候选词。

```
ts_stat(_sqlquery_ text, [ `_weights_` `text`, ]
        OUT _word_ text, OUT _ndoc_ integer,
        OUT _nentry_ integer) returns setof record
```

`_sqlquery_` 是一个包含返回单独 `tsvector` 列的SQL查询的文本值。 `ts_stat` 执行查询并返回包含 `tsvector` 数据的各个不同的语义（词）的统计。返回的列：

- `_word_` `text` — 一个词的值
- `_ndoc_` `integer` —这个词出现的文档编号（ `tsvector` s）
- `_nentry_` `integer` —这个词出现的总数

如果提供 `_weights_` ，仅仅计算这些权重之一。

例如，在一个文档集合中查找十个最常用的单词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的，但是只计算权重weight `A` 或者 `B` 的单词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. 解析器

文本搜索分析器负责分离原文档文本为标记并且标识每个记号的类型，这里可能的类型集由解析器本身定义。 注意一个解析器并不修改文本—它只是确定合理的单词边界。因为这个限制范围， 为特定应用定制的分析器比自定义字典需要的更少。目前PostgreSQL提供了只有一个内置的解析器， 这已被用于一个广泛的应用中。

内置分析器命名`pg_catalog.default`。它识别23种标记类型，显示在Table 12-1中。

Table 12-1. 缺省分析器的标记类型

Alias	Description	Example
asciiword	Word, all ASCII letters	elephant
word	Word, all letters	mañana
numword	Word, letters and digits	beta1
asciihword	Hyphenated word, all ASCII	up-to-date
hword	Hyphenated word, all letters	lógico-matemática
numhword	Hyphenated word, letters and digits	postgresql-beta1
hword_asciipart	Hyphenated word part, all ASCII	postgresql in the context postgresql-beta1
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática
hword_numpart	Hyphenated word part, letters and digits	beta1 in the context postgresql-beta1
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html , in the context of a URL
file	File or path name	/usr/local/foo.txt , if not within a URL
sfloat	Scientific notation	-1.234e56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	<a href="dictionaries.html">
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

> **Note:** 注意：一个"字母"的语法分析器的概念是由数据库的区域设置决定的，特别是 `lc_ctype`。 只包含基本ASCII字母的词作为一个单独的标记类型被报告，因为区分他们有时候是有用的。 大多数欧洲语言，标记类型 `word` 和 `asciiword` 应该一视同仁。
>
> `email` 不支持由RFC 5322定义的所有有效的电子邮件字符。具体来说， 唯一的非字母数字字符支持电子邮件用户名有句号，破折号和下划线。

对于分析器从文本的同一块产生重叠的标记是可能的。作为一个例子， 一个连字符的单词将作为整个单词和每个组件被报道：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

这种行为是可取的，因为它允许为整个复合词和组件进行搜索。这里是另一个很好的例子:

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. 词典

词典用于删除那些不在搜索范围内的词（屏蔽词），并且为了规范化，将匹配同一个词的不同形式。 一个成功的规范化的词叫词位。除了提高检索质量外，屏蔽词的规范化和删除可以减少文档 `tsvector` 形式的大小， 从而提高性能。规范化并不总是有语言学意义，通常取决于应用程序的环境。

一些规范化的例子：

- 语言的-Ispell词典尽量减少输入字的正规形式；词干词典去掉词尾
- URL位置可以使等效URL匹配被规范化：
    - http://www.pgsql.ru/db/mw/index.html
    - http://www.pgsql.ru/db/mw/
    - http://www.pgsql.ru/db/../db/mw/index.html
- 颜色名称可以由他们的十六进制值替换，比如：`red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- 如果索引数字，我们可以删除一些小数位数，减少可能数字的范围，例如如果保留小数点后两位小数， 则*3.14_159265359,_3.14_15926*将归一化为一样的*_3.14*

字典是一个程序，它接受标记作为输入和返回：

- 词条数组如果输入标记是已知的词典（注意，一个标记可以产生一个以上的词）
- 用 `TSL_FILTER` 标志设置的单词，与被传递到随后的词典的新的标记代替原来的（称这是过滤词典）
- 如果词典认为标记是空数组，但它是一个屏蔽词。
- 如果词典不能识别输入标记，则为 `空`

PostgreSQL提供了多种语言的预定义字典。也有几个预定义的模板，可用于创建自定义参数的新词典。 每个预定义的字典模板描述如下。如果没有现成的模板是合适的，它可以创建一个新的； 参见PostgreSQL发布的 `contrib/` 部分例子

文本搜索配置将解析器和处理解析器输出标记绑定在一起。为了每个标记类型， 返回解析器，单独的词典列表通过配置指定。当标记类型是由解析器发现时， 列表中的每个字典依次查阅，直到一些词典作为一个已知的单词识别它。如果它被确定为一个屏蔽词， 或者如果没有词典识别标记，它将被丢弃，并且没有索引或搜索。通常，返回一个非- `空` 输出的第一个词典将决定结果， 并且不查阅任何剩余的词典；但过滤词典可以替换带有修饰词的给定词，然后被传递给后继词典。

配置一个字典列表的一般规则是放在第一个最窄的，最具体的词典中，然后是更一般的词典， 整理一个非常普遍的词典，像Snowball词干或 `simple` 可以识别一切。例如， 一个天文学的特定搜索（ `astro_en` 配置）可以将标记类型 `asciiword` （ASCII字）绑定到天文术语的同义词词典， 一般英语词典和Snowball 英文词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

过滤词典可以放置在列表中的任何地方，除了在结束的地方会是无用的。 过滤词典部分规范化词以简化后继词典的任务是非常有用的。 例如，过滤词典可以用来从重音字母中删除重音，按照unaccent模块执行。

12.6.1. 屏蔽词

屏蔽词是很常见的词，出现在几乎每一个文档中，并且没有区分值。因此，他们可以在全文搜索的环境中被忽视的。 例如，每个英文文本包含像 `a` 和 `the` 的单词，因此它们在索引中存储无效。然而，屏蔽词影响在 `tsvector` 中的位置， 这反过来也影响相关度：

```
SELECT to_tsvector('english','in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

丢失位置1,2,4是因为屏蔽词。带有和没有屏蔽词的文档排序计算是完全不同的：

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list &
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

如何处理屏蔽词，它是由特定词典决定的。例如， `ispell` 词典首先规范词，然后查看屏蔽词列表， 而 `Snowball` 词干首先检查屏蔽词列表。这个不同操作的原因是为了减少噪音。

12.6.2. Simple 词典

`simple` 字典模板通过转换输入标记为小写字母进行，并且屏蔽词文件前检查它。 如果在文档中找到并返回空数组，则丢弃这个标记。如果没有，单词的小写字母形式作为归一化的词返回。 另外，词典可以为报告未识别的非屏蔽词进行配置，允许将它们传递到列表中的后继词典中。

这有使用 `simple` 模板的词典定义的例子：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

在这里， `english` 是一种屏蔽词文件的基础名称。文档的全名为 `$SHAREDIR/tsearch_data/english.stop` ， 这里的 `$SHAREDIR` 是PostgreSQL安装的共享数据目录，经常使用 `/usr/local/share/postgresql` （如果你不确定， 则使用 `pg_config --sharedir` 来决定）。文档格式是一个简单的单词列表，每行一个。忽略空白行和空格， 并且大写字母转换成小写字母，但对文档内容没有其他的处理方式。

现在我们可以测试我们的词典：

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

如果没在屏蔽词文件中找到，我们也可以选择返回 `NULL` ， 而不是小写字母单词。 这种行为是通过设置字典的 `Accept` 参数为 `false` 选择的。继续例子：

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

随着缺省设置 `Accept` = `true` ， 它把 `simple` 词典放在词典列表末尾的时候是很有用的， 因为它不会传递任何标记给后继词典。相反，当至少有一个后继词典时， `Accept` = `false` 是唯一有用的。

Caution

词典大部分类型依赖于配置文档，如屏蔽词文件。这些文件必须存储在UTF-8编码中。 当他们读到服务器中，如果是不同的，他们将被转化为实际的数据库编码。

Caution

通常情况下，当它在会话中第一次使用时，数据库会话将只读一次词典的配置文档， 如果你修改一个配置文档，想强制现有会话获取新的内容，则在词典中使用命令 `ALTER TEXT SEARCH DICTIONARY` 。 这是一个"虚拟"的更新，实际上并没有改变任何参数值。

12.6.3. 同义词词典

这个字典模板用于创建替代词和同义词的词典。不支持短语（使用同义词库模板（节Section 12.6.4）。 一个同义词词典可以用来克服语言上的问题，例如，防止英语词干词典使单词"Paris"变成"pari"。 这足以在同义词词典中有 `Paris paris` 行并且放在 `english_stem` 词典之前。比如：

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

`synonym` 模版要求的唯一的参数是 `SYNONYMS` ， 这是它的配置文件的基础名称—上面例子中 `my_synonyms` 。 文件的全名为 `$SHAREDIR/tsearch_data/my_synonyms.syn` （ `$SHAREDIR` 是PostgreSQL安装的共享数据目录）。 文件格式是每一行的每个字被取代，带有这个词的同义词，用空格分隔。忽略空白行和空格。

`synonym` 模版也有一个可选的参数 `CaseSensitive` ，缺省是 `false` 。 当 `CaseSensitive` 是 `false` 时， 同义词文件中的词转换成小写字母，正如输入标记。 比较而言，当它是 `true` 时，词语和标记不转换成小写字母

星号（ `*` ）可以被放置在配置文件中的同义词结尾。这表明，同义词是一个前缀。 当在 `to_tsvector()` 中使用记录时，忽略星号。但当它被用在 `to_tsquery()` 中时， 结果将是带前缀匹配标记的查询记录（参见节Section 12.3.2）。例如，假设我们在 `$SHAREDIR/tsearch_data/synonym_sample.syn` 中有这些记录。

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

然后我们将得到这些结果：

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. 同义词词典库

同义词词库（有时简称TZ）是一个单词的组合，包括单词和短语的关系信息， 比如，更广泛术语（BT），更窄的术语（NT），首选术语，非优先术语，相关术语等。

基本上同义词词库通过一个首选的术语替换所有非优先术语， 另外，也保留索引的原术语。同义词词典PostgreSQL的当前实现是带有附加短语支持的同义词词典的扩展。同义词词典需要下列格式的配置文件：

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

冒号（ `:` ）符号作为短语和其替代物之间的分隔符。

同义词词典检查短语匹配之前使用一个子词典（这是在字典的配置中指定）规范输入文本。它选择一个子词典是可能的。如果子词典无法识别单词，报告一个错误。在这种情况下， 你应该删除这个词或训练子词典。你可以在一个索引字跳过应用子词典的开头放一个星号（ `*` ）， 但是所有简单的词必须是子词典已知的。

如果有多个短语匹配输入，同义词词典选择最长的匹配。并且使用最后一个定义分离关系。

通过子词典识别的具体屏蔽词不能被指定；而使用 `?` 标记任何屏蔽词出现的位置。 例如，假设 `a` 和 `the` 是依据子词典的屏蔽词：

```
? one ? two : swsw
```

匹配 `a one the two` 和 `the one a two` ；两者都会被 `swsw` 替代。

由于同义词词典有能力识别短语，它必须记住其状态并且与分析器交互。同义词词典使用这些任务检查它是否应该处理下一个词， 或停止积累。同义词词典必须小心配置。例如，如果字典词库分配只处理 `asciiword` 标记，那么像 `one 7` 的同义词词典定义将不工作， 因为标记类型 `uint` 不分配给同义词词典。

Caution
索引中使用词典，同义词词典的任何参数变化都需要重新索引。对于大多数其他词典类型， 小的变化，比如添加或去除屏蔽词不强迫重新索引。

12.6.4.1. 同义词词典配置

使用 `thesaurus` 模板定义一个新的同义词词库。比如:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

这里：

- `thesaurus_simple` 是新词典的名称。
- `mythesaurus` 是同义词配置文件的基础名称。 （全名为 `$SHAREDIR/tsearch_data/mythesaurus.ths` ， 这里 `$SHAREDIR` 是安装的共享数据目录）
- `pg_catalog.english_stem` 是用于词规范化的子词典（这的Snowball英文词干）。 注意，子词典将有自己的配置（例如，屏蔽词）， 不显示在这里。

现在它在配置中可能将同义词词典 `thesaurus_simple` 绑定到所需的标记类型中，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. 同义词词典例子

考虑一个简单的天文词典 `thesaurus_astro`，其中包含了一些天文组合词：

```
supernovae stars : sn
crab nebulae : crab
```

下面我们创建一个词典并且绑定标记类型的一些天文词库和英文词干分析器:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

现在我们可以看到它是如何工作的。 `ts_lexize` 对测试一个词库没有很大帮助，因为它把输入作为一个标记。 相反，我们可以使用 `plainto_tsquery` 和 `to_tsvector`，将它们的输入字符串分离成多个标记：

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

原则上，如果你引用参数，可以使用 `to_tsquery`：

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

注意 `supernova star` 与 `supernovae stars` 在 `thesaurus_astro` 中匹配，因为我们在词典的定义中指定了 `english_stem` 词干分析器。 词干分析器删除 `e` 和 `s` 。

为了索引原句以及替代词，只是将它包括在定义的右边部分：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell词典

Ispell词典模版支持形态学的词典，它可以将一个单词的许多不同的语言形式标准化为一个词。 例如，英语Ispell词典可以匹配所有词尾变化和搜索词 `bank` 的组合， 例如， `banking` , `banked` , `banks` , `banks'` 和 `bank's` 。

标准的PostgreSQL发布不包括任何Ispell配置文件。大量的语言字典可以从Ispell获得。 同时，— MySpell (OO < 2.0.1) 和Hunspell(OO >= 2.0.2)支持一些更现代的词典文件格式。 大的词典列表在OpenOffice Wiki中可用。

使用内置的 `ispell` 模板创建Ispell 词典，并指定几个参数：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

这里， `DictFile` , `AffFile` 和 `StopWords` 指定词典基础的名字，词缀，和屏蔽词文件。 屏蔽词文件具有和上面解释的 `simple` 词典类型相同的格式。其它文件的格式不在这里指定， 但可以从上面提到的网站获取。

Ispell词典通常识别有限的一组词，所以他们应该遵循另一个更广泛的词典； 例如，一个Snowball词典，它可以识别一切。

Ispell词典支持分裂复合词；一个有用的功能。请注意， 词缀文件应使用 `compound words controlled` 语句指定一个特殊标记， 标记可以参与复合信息的词典单词：

```
compoundwords  controlled z
```

这有一些Norwegian语言的例子：

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

> **Note:** 注意：MySpell不支持复合词。Hunspell对复合词有复杂支持。 目前，PostgreSQL只实现了Hunspell的基本复合词操作。

12.6.6. Snowball词典

Snowball词典模板是基于Martin Porter的项目，他是英语语言的著名的Porter的词干提取算法的发明者。 现在Snowball提供了许多语言的词干提取算法（更多信息请见Snowball site）。每个算法都知道如何改变词到基础，或词根，或其语言拼写的常见变异形式。 一个Snowball词典需要 `language` 参数标识要使用的词干，并且可以指定一个删除词的列表的 `stopword` 文件名。 （PostgreSQL的标准的屏蔽词列表也由Snowball项目提供）例如，有一个等价的内置定义。

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

屏蔽词的文件格式和已经解释过的一样。

一个Snowball词典可以识别一切，是否能够简化字，所以它应该放在词典列表的末尾。 它放在任何其他的词典之前都是无用的，因为一个标记将不会经过它到下一个词典。

12.7. 配置实例

文本搜索配置指定所有选项将文档转换成一个 `tsvector` ：使用解析器将文本分解为标记， 并且使用词典将每个标记转换为词。 `to_tsvector` 或者 `to_tsquery` 的每一次调用需要一个文本搜索配置来执行处理。 如果一个明确的配置参数被省略，则配置参数default_text_search_config指定默认配置的名称， 它是通过一个文本搜索函数使用的。它可以在 `postgresql.conf` 中设置，或使用 `SET` 命令设置一个独立会话。

有几个预定义的文本搜索配置是可用的，并且您可以很容易的创建自定义的配置。 为了方便文本搜索对象的管理，一组SQL命令是可用的， 有几个psql命令可以显示有关文本搜索对象的信息（节Section 12.10）。

作为一个例子，我们将创建一个配置 `pg` ，通过复制内置 `english` 配置启动：

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

我们将使用PostgreSQL特定的同义词列表并将其存储在 `$SHAREDIR/tsearch_data/pg_dict.syn` 中。 文件内容看起来像：

```
postgres    pg
pgsql       pg
postgresql  pg
```

我们定义这样的同义词词典：

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

接下来我们注册Ispell词典 `english_ispell` ，它有自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

现在我们可以在配置 `pg` 中建立词汇映射：

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

我们没有选择索引或搜索一些内置配置处理的标记类型：

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

现在我们测试我们的配置：

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一步是使用新的配置设置会话，这是在 `public` 模式中建立的：

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. 测试和调试文本搜索

一个自定义文本搜索配置的行为很容易变得混乱。在本节中描述的函数对测试文本搜索对象是有用的。 你可以测试一个完整的配置，或分别测试分析器和词典。

12.8.1. 配置测试

函数 `ts_debug` 允许简单测试文本搜索配置。

```
ts_debug([ `_config_` `regconfig`, ] _document_ text,
         OUT _alias_ text,
         OUT _description_ text,
         OUT _token_ text,
         OUT _dictionaries_ regdictionary[],
         OUT _dictionary_ regdictionary,
         OUT _lexemes_ text[])
         returns setof record
```

`ts_debug` 显示关于通过解析器产生的和通过配置词典处理的 `_document_` 的每个标记的信息。如果忽略参数，它使用通过 `_config_` 或者 `default_text_search_config` 指定的配置。

`ts_debug` 返回通过文本解析器标识的每个标记的每一行，返回的列是：

- `_alias_` `text` —标记类型的别名
- `_description_` `text` —标记类型描述
- `_token_` `text` —标记文本
- `_dictionaries_` `regdictionary[]` —通过配置为这个标记类型选定的词典
- `_dictionary_` `regdictionary` —词典公认的标记，如果不这样，则为 `空` 。
- `_lexemes_` `text[]` — 公认标记的词典产生的词（s），或者如果不做则为 `NULL` ；空数组（ `{}` ）意味着它是公认的屏蔽词。

一个简单例子：

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

一个更广泛的例子，我们首先用英语创建一个 `public.english` 配置和Ispell词典：

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |   dictionary
-----------+-----------------+-------------+-------------------------------+-------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} | english_ispe
 blank     | Space symbols   |             | {}                            |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} | english_ispe
 blank     | Space symbols   |             | {}                            |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} | english_stem
```

在这个例子中， `Brightest` 是由解析器作为 `ASCII词` 来标识（别名 `asciiword` ）。为这个标识类型，词典列表是 `english_ispell` 和 `english_stem` 。这个词通过 `english_ispell` 标识，归纳它为名词 `bright` 。词 `supernovaes` 于 `english_ispell` 词典是未知的，所以它传递给下一个词典， 幸运的是，是公认的（事实上， `english_stem` 是一个识别一切的Snowball词典；这就是为什么它被放置在词典列表末尾的原因）。

词 `The` 是由 `english_ispell` 词典被公认为屏蔽词（节Section 12.6.1），不会被索引。 空间也被丢弃，因为该配置根本没有为它们提供词典。

你可以通过明确指定你想要查看的列减少输出的宽度：

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. 解析器测试

下列函数允许直接测试文本搜索解析器。

```
ts_parse(_parser_name_ text, _document_ text,
         OUT _tokid_ integer, OUT _token_ text) returns setof record
ts_parse(_parser_oid_ oid, _document_ text,
         OUT _tokid_ integer, OUT _token_ text) returns setof record
```

`ts_parse` 解析给定的 `_document_` 并返回一系列的记录，每一个标记通过解析而产生。 每个记录包括 `tokid` 显示已分配的标记类型，并且 `token` 是标记的文本。比如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(_parser_name_ text, OUT _tokid_ integer,
              OUT _alias_ text, OUT _description_ text) returns setof record
ts_token_type(_parser_oid_ oid, OUT _tokid_ integer,
              OUT _alias_ text, OUT _description_ text) returns setof record
```

`ts_token_type` 返回一个表，这个表描述了每种可以识别的指定分析器标记类型。 每个标记类型，该表给出了整数 `tokid` ，解析器用于标记那个类型标记， `alias` 命名配置命令的标记类型， 并且简称 `description` 。比如：

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. 词典测试

`ts_lexize`函数有易于进行词典测试。

```
ts_lexize(_dict_ regdictionary, _token_ text) returns text[]
```

如果输入 `_token_` 为词典已知的，那么 `ts_lexize` 返回词的数组，如果这个token对词典是已知的， 但它是一个屏蔽词，则返回空数组。如果它是一个未知的词则返回 `NULL` 。

比如：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Note:** `ts_lexize`函数需要单一标记，没有文本。这是一种引起混淆的情况：
>
> ```
> SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```
>
> 同义词词典`thesaurus_astro`确实知道短语`supernovae stars`，但`ts_lexize`失败了，因为它不解析输入文本，而是把它作为一个单一标记。使用`plainto_tsquery`或者`to_tsvector`测试同义词词典，例如：
>
> ```
> SELECT plainto_tsquery('supernovae stars');
>  plainto_tsquery
> -----------------
>  'sn'
> ```

12.9. GiST和GIN索引类型

有两种类型的索引可以用于加快全文搜索。注意全文检索不一定非要使用索引。 但是在规则基础上搜索列的情况下，索引往往是可取的。

`CREATE INDEX` `_name_` ON `_table_` USING gist(`_column_`);

创建以GiST（通用搜索树）为基础的索引， `_column_` 可以是 `tsvector` or `tsquery` 类型。

`CREATE INDEX` `_name_` ON `_table_` USING gin(`_column_`);

创建以GIN（基因倒排索引）为基础的索引， `_column_` 必须是 `tsvector` 类型。

在两个索引类型之间有着巨大的性能差异，因此了解它们的特性是很重要的。

GiST索引是有损耗的，这意味着该索引可能会产生错误的匹配， 并且有必要检查实际的表行消除这种错误匹配（PostgreSQL需要时自动执行）。 GiST索引是有损耗的，因为每个文档在索引中通过一个固定长度的标签进行表示。 它是通过散列每个单词到一个n位的字符串的单一的点产生，所有这些位OR-ed一起产生一个n位的文件标签。 当两个单词散列到相同点的位置，将有一个错误匹配。如果查询中的所有单词匹配（真实的或错误的）， 则必须检索表行查看匹配是否是正确的。

数据丢失导致了性能下降，由于表记录的不必要的获取，产生了错误的匹配。 由于随机访问表记录是缓慢的，这限制了GiST索引的效能。错误匹配的可能性取决于几个因素， 特别是独特词的数量，所以推荐使用词典来降低这些数量。

GIN索引并没有损耗标准查询，但它们的性能取决于对数独特的单词数。 （然而，GIN索引只存储 `tsvector` 值的字（词）， 而不是它们的权重标签。因此， 当使用涉及权重的查询时，需要复查一个表行。）

在选择要使用的索引类型时，GiST或者GIN考虑这些性能上的差异：

- GIN索引查找比GiST快约三倍
- GIN索引建立比GIST需要大约三倍的时间。
- GIN索引更新比GiST索引速度慢，但如果快速更新支持无效，则慢了大约10倍（详情请见节Section 57.3.1）
- GIN索引比GiST索引大两到三倍

一般来说，GIN索引对静态数据是最好的，因为查找速度很快。对于动态数据， GiST索引更新比较快。具体而言，GiST索引非常适合动态数据，并且如果独特的字（词）在100,000以下， 则比较快，而GIN索引将处理100,000+词汇，但是更新比较慢。

请注意，GIN索引编译时间通常可以通过增加maintenance_work_mem改进， 而GiST索引编译时间对参数不敏感。

大集合的分区以及GiST和GIN索引的合理使用允许非常快速的搜索与在线升级的实现。 分区可以在数据库级别使用表继承，或者在服务器发布文档并且使用dblink模块采集搜索结果。 后者是可能的，因为相关函数只使用本地信息。

12.10. psql支持

文本搜索配置对象的信息可以使用一组命令从psql中获得：

```
\dF{d,p,t}[+] [PATTERN]
```

一个可选的`+`产生更多细节。

可选的参数`PATTERN`可以是一个文本搜索对象的名称，随意的模式匹配。 如果`PATTERN`被忽略，则显示所有可见对象的信息。 `PATTERN`可以是一个正则表达式， 并且可以提供模式的独立形式和对象名称。下面的例子说明了这些：

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name         | Description
--------+--------------+-------------
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name         | Description
----------+---------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

可用命令是：

`\dF[+] [PATTERN]`

罗列文本搜索配置（增加`+`获取更多细节）：

```
=&gt; \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+------------------------------------
 pg_catalog | russian | configuration for russian language

=&gt; \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

`\dFd[+] [PATTERN]`

罗列文本搜索词典（增加 `+` 获取更多细节）。

```
=&gt; \dFd
                             List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------+-----------------------------------------------------------
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

`\dFp[+] [PATTERN]`

罗列文本搜索分析器（增加 `+` 获取更多的细节）。

```
=&gt; \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=&gt; \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

`\dFt[+] [PATTERN]`

罗列文本搜索模板（增加 `+` 获取更多的细节）。

```
=&gt; \dFt
                           List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. 限制

PostgreSQL的文本搜索功能当前限制是：

- 每个词的长度必须小于2K字节
- `tsvector` （词+位置）的长度必须小于1兆字节
- 词的数量必须小于2⁶⁴
- `tsvector` 的位置值必须大于0，不能超过16,383
- 每词不超过256位置
- 在 `tsquery` 中节点的数目（词+运算符）必须小于32768

相比之下，PostgreSQL 8.1文档包含10441个唯一的字，共335420个字， 并且最频繁的词"postgresql"在655个文档被提到6127次。

另一个例子— PostgreSQL邮件列表档案包含910989个唯一的字与461，020消息中的57491343个词。

12.12. 来自8.3之前文本搜索的迁移

为文本搜索使用tsearch2模块的应用将需要内置功能的一些调整。

- 一些函数已被重命名或在其参数列表有小的调整，并且他们现在都在 `pg_catalog` 模式中， 而在以前的安装中都是在 `public` 或另一种非系统模式中。有一个新的tsearch2版本， 它提供了兼容层来解决这方面的问题。
- 当从8.3之前数据库加载pg*dump*输出时，必须_抑制旧的tsearch2函数和其他对象。 而他们中的许多不会加载，一些会导致问题。一个简单处理方法就是恢复转储前加载新的tsearch2模块； 然后阻塞被加载的旧对象。
- 文本搜索配置设置现在完全不同。不是手动插入行到配置表，搜索是通过本章节前面显示的专门的SQL命令配置。 没有自动支持8.3转换现有的自定义配置；你可以在这里自己定义。
- 大多数类型的词典依靠一些外部的数据库配置文件。这些与8.3之前用法兼容，但注意以下的差异：
  - 配置文件现在必须放在一个单一指定的目录（ `$SHAREDIR/tsearch_data` ）中， 必须有一个特定的扩展取决于文件的类型，如先前在各种词典类型的描述中指出的。 这个限制被添加到安全问题中。
  - 无论使用什么数据库编码，配置文件必须以UTF-8编码。
  - 在词库的配置文件中，屏蔽词必须用 `?` 标记。

Chapter 13. 并发控制

Table of Contents

本章描述PostgreSQL数据库系统在多个会话试图同时访问同一数据时的表现。并发控制的目标是为所有会话提供高效的访问，同时还要维护严格的数据完整性。每个数据库应用开发人员都应该熟悉本章讨论的话题。

13.1. 介绍

PostgreSQL为开发者提供了丰富的对数据并发访问进行管理的工具。 在内部，PostgreSQL利用多版本并发控制(MVCC)来维护数据的一致性。 这就意味着当检索数据时，每个事务看到的都只是一小段时间之前的数据快照(一个数据库版本)， 而不是数据的当前状态。这样，如果对每个数据库会话进行事务隔离，就可以避免一个事务看到其它并发事务的更新而导致不一致的数据。MVCC通过避开传统数据库系统锁定的方法，最大限度地减少锁竞争以允许合理的多用户环境中的性能。

使用多版本并发控制比锁定模型的主要优点是在MVCC里， 对检索(读)数据的锁请求与写数据的锁请求不冲突，所以读不会阻塞写，而写也从不阻塞读。甚至当通过创新的序列化快照隔离 (SSI)级别提供事务隔离的严格等级时， PostgreSQL维持这样的保证。

在PostgreSQL里也有表和行级别的锁定机制，用于给那些无法轻松接受MVCC行为的应用。不过，恰当地使用MVCC总会提供比锁更好的性能。另外，由应用定义的咨询锁提供了一个获得不依赖于单独事务的锁的机制。

13.2. 事务隔离

SQL标准定义了四个级别的事务隔离。最严格的是串行化，它是通过标准来定义的，也就是说， 保证一组可序列化事务的并发执行以产生同样顺序依次运行它们的同一效果。 其他三个层次是通过现象术语被定义，导致并发事务之间的相互作用，这不应该发生在每个级别中。标准定义归因于序列化的定义，这些现象不可能在这一水平上（这毫不奇怪--如果事务的影响必须与已运行的一个保持一致，你怎么能看到通过相互作用引起的现象呢？

各个级别不希望发生的现象是：

脏读

一个事务读取了另一个未提交事务写入的数据。

不可重复读

一个事务重新读取前面读取过的数据，发现该数据已经被另一个已提交事务修改。

幻读

一个事务重新执行一个查询，返回一套符合查询条件的行，发现这些行因为其它最近提交的事务而发生了改变。

这四种隔离级别和对应的行为在表Table 13-1里描述。

Table 13-1. **标准SQL事务隔离级别**

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

在PostgreSQL里，你可以请求四种可能的事务隔离级别中的任意一种。但是在内部， 实际上只有三种独立的隔离级别，分别对应读已提交，可重复读和可串行化。如果你选择了读未提交的级别， 实际上你用的是读已提交，在重复读的PostgreSQL执行时，幻读是不可能的，所以实际的隔离级别可能比你选择的更严格。这是 SQL 标准允许的：四种隔离级别只定义了哪种现像不能发生， 但是没有定义那种现像一定发生。PostgreSQL只提供两种隔离级别的原因是， 这是把标准的隔离级别与多版本并发控制架构映射相关的唯一合理方法。可用的隔离级别的行为在下面小节里描述。

要设置一个事务的隔离级别，使用SET TRANSACTION命令。

> **Important:** 一些PostgreSQL数据类型和函数关于事务行为有特定的规则。 尤其是，序列变化（因此列数通过 `serial` 声明）对于所有其他的事务是立即可见的， 如果事务改变终止，则不进行回退。参见Section 9.16和Section 8.1.4。

13.2.1. 读已提交隔离级别

读已提交是PostgreSQL里的缺省隔离级别。当一个事务运行在这个隔离级别时， `SELECT` 查询(没有 `FOR UPDATE/SHARE` 子句)只能看到查询开始之前已提交的数据而无法看到未提交的数据或者在查询执行期间其它事务已提交的数据 。实际上， `SELECT` 查询看到一个在查询开始运行的瞬间该数据库的一个快照。 不过， `SELECT` 看得见其自身所在事务中前面更新执行结果。即使它们尚未提交。请注意， 在同一个事务里两个相邻的 `SELECT` 命令可能看到不同的快照，因为其它事务会在第一个 `SELECT` 执行期间提交。

`UPDATE` ， `DELETE` ， `SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 命令在搜索目标行时的行为和 `SELECT` 一样： 它们只能找到在命令开始的时候已经提交的行。不过， 这样的目标行在被找到的时候可能已经被其它并发事务更新、删除、锁住。在这种情况下， 即将进行的更新将等待第一个事务提交或者回滚(如果它还在处理)。如果第一个事务回滚， 那么它的作用将被忽略，而第二个事务将继续更新最初发现的行。如果第一个事务提交， 那么如果第一个事务删除了该行，则第二个事务将忽略该行， 否则它将试图在该行的已更新的版本上施加它的操作。系统将重新计算命令搜索条件(`WHERE` 子句)， 看看该行已更新的版本是否仍然符合搜索条件。如果符合，则第二个事务从该行的已更新版本开始继续其操作。 如果是 `SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 则意味着把已更新的行版本锁住并返回给客户端。

因为上面的规则，正在更新的命令可能会看到不一致的快照： 它们可以看到影响它们更新的并发命令的效果，但是却看不到那些命令对数据库里其它行的作用。 这样的行为令读已提交模式不适合用于哪种涉及复杂搜索条件的命令。不过，它对于简单的情况而言是正确的。 比如，假设我们用类似下面这样的命令更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个并发事务试图同时修改帐号12345的余额，那我们很明显希望第二个事务是从已更新过的行版本上进行更新。 因为每个命令只是影响一个已经决定了的行，因此让它看到更新后的版本不会导致任何不一致的问题。

更复杂的用法可以在读已提交模式下产生不需要的结果。比如，考虑 `DELETE` 命令数据操作 通过另外一个命令的限制标准中被添加或者删除等，假设 `website` 是 `website.hits` 等同于 `9` 和 `10` 的两行表格。

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

`DELETE` 不会产生影响，即使在 `UPDATE` 之前和之后有 `website.hits = 10` 。 这发生是因为先前更新的行值 `9` 被忽略，并且当 `UPDATE` 完成而且 `DELETE` 获得锁时， 新的行值不再是 `10` 而是 `11` ， 它不再符合标准。

因为在读已提交模式里，每个新的命令都是从一个新的快照开始的， 而这个快照包含所有到该时刻为止已提交的事务， 因此同一事务中后面的命令将看到任何已提交的其它事务的效果。 这里关心的问题是在单个命令里是否看到数据库里绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快，使用简单。 不过，对于做复杂查询和更新的应用， 可能需要保证数据库有比读已提交模式更加严格的一致性视图。

13.2.2. 可重复读隔离级别

可重复读隔离级别仅仅看到事务开始之前提交的数据，它不能看到在并发事务执行期间未提交的数据和已提交的改变。 （然而，查询看到在自身事务中执行的先前更新的效果，即使它们没有被提交）。比为这一隔离级别的SQL标准需求来说，这是一个更强烈的保证。 避免所有在Table 13-1描述的现象。正如上述所提及的，这是通过标准允许的， 这仅仅描述必须提供的每个隔离级别的最低限度保护。

这个级别和读已提交级别是不一样的。重复读事务中的查询看到的是事务开始时的快照， 而不是该事务内部当前查询开始时的快照，这样， 同一个事务内部后面的 `SELECT` 命令总是看到同样的数据等，它们没有看到通过 自身事务开始之后提及的其他事务做出的改变。

使用这个级别的应用必须准备好重试事务，因为串行化失败。

`UPDATE` ， `DELETE` ， `SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 在搜索目标行时的行为和 `SELECT` 一样： 它们将只寻找在事务开始的时候已经提交的目标行。但是， 这样的目标行在被发现的时候可能已经被另外一个并发的事务更新、删除、锁住。在这种情况下， 可串行化的事务将等待第一个正在更新的事务提交或者回滚(如果它仍然在处理中)。如果第一个事务回滚， 那么它的影响将被忽略，而这个可串行化的就可以继续更新它最初发现的行。 但是如果第一个事务被提交了(并且实际上更新或者删除了该行，而不只是锁住它)那么可串行化事务将回滚， 并返回下面信息：

```
ERROR:  could not serialize access due to concurrent update
```

因为一个可串行化的事务在开始之后不能更改或者锁住被其它事务更改过的行。

当应用收到这样的错误信息时，它应该退出当前的事务然后从新开始进行整个事务。第二次运行时， 该事务看到的快照将包含前一次已提交的修改，所以不会有逻辑冲突。

请注意只有更新事务才需要重试，只读事务从来没有串行化冲突。

可重复读事务级别提供了严格的保证：每个事务都看到一个完全一致的数据库视图。然而，这种观点也不一定总是与（一次一个）同一级别的并发事务连续执行一致。 例如，即使在这个级别上的一个只读事务可以看到控制记录更新显示一批已经完成，但 不能看到一批逻辑部分的详细记录， 因为它读取较早版本的控制记录。如果不仔细使用显式锁来阻止冲突事务，通过运行在这个隔离级别上的事务尝试执行业务规则是不能正常工作的。

> **Note:** PostgreSQL9.1版本之前，为序列化事务隔离级别的请求提供完全相同的描述。为保留传统的串行化行为，现在要求可重复读。

13.2.3. 可串行化隔离级别

可串行化级别提供最严格的事务隔离。这个级别为所有已提交事务模拟串行的事务执行， 就好像事务将被一个接着一个那样串行(而不是并行)的执行。不过，正如可重复读隔离级别一样， 使用这个级别的应用必须准备在串行化失败的时候重新启动事务。 事实上，该隔离级别和可重复读希望的完全一样， 它只是监视这些条件，以所有事务的可能的序列不一致的（一次一个）的方式执行并行的可序列化事务执行的行为。 这种监测不引入任何阻止可重复读出现的行为，但有一些开销的监测，检测条件这可能会导致序列化异常 将触发序列化失败。

举例来说，假设一个表 `mytab` ， 最初包含:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

假设可串行化事务 A 计算:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

然后把结果(30)作为 `value` 字段值插入到表中，并令新行的 `class` `= 2` 。同时， 另一个并发的可串行化的事务B进行下面计算

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

然后把结果(300)作为 `class` `= 1` 字段值插入到表中。 然后两个事务都提交。如果事务都在可重复读隔离级别上运行， 两者都不允许提交；但是因为 没有执行一致性结果的序列顺序，使用可串行化事务将允许一个事务被提交，并且回滚到该消息的其他块中。

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

这是因为如果 A 在 B 之前执行，B 应该计算出总和 330， 而不是300， 如果B在A之前执行，那么 A 计算出的总和也会不同。

当依赖于可串行化事务阻止异常时，来自永久用户表读取的任何数据不被认为是有效的，直到事务读取的成功提交为止。 这对于只读事务是真的，除了在可延期的只读事务中的数据读是有效的。 因为这样一个事务等待直到它可以在开始读取任何数据之前获得一个快照保证这些问题是自由的。 在所有其他情况下，应用不依赖于结果读，期间事务之后被停止；相反，他们应该重启事务直到成功为止。

为了保证PostgreSQL真正可串行化使用谓词锁定。 这意味着当写对于并发事务的先前读结果有重大影响时，它使锁决定首先运行。 在PostgreSQL这些锁不造成任何阻塞，因此可以不导致僵局。 它们被用来识别和标记并发序列化事务中的依赖关系，其中一定的组合可导致序列化异常。 相反，读已提交或者可重复读取的事务要确保数据的一致性可能需要获取整个表锁， 它可以阻止其他尝试使用该表的用户，也可以使用 `SELECT FOR UPDATE` 或者 `SELECT FOR SHARE` ，这不仅可以阻止其他事务而且可能导致磁盘访问。

PostgreSQL中的谓词锁，像其他大多数数据库系统一样， 基于通过事务实际访问的数据，这些显示在 `pg_locks` 系统视图中，并带有 `SIReadLock` 的 `模式` 。 查询执行期间特定的锁的获得将取决于使用的查询计划。以及事务进程防止用于跟踪锁定的内存耗尽期间的多个细粒度锁（例如，元组锁）可以组合成较少的粗粒度的锁（例如，页锁）。 `只读` 事务可以在完成之前释放SIRead锁，如果它检测到没有冲突仍然发生，这可能会导致一系列的异常。事实上， `只读` 事务会经常建立启动事实，并且避免采取任何谓词锁。如果你明确要求 `SERIALIZABLE READ ONLY DEFERRABLE` 事务，这将阻塞直到它可以建立这一事实。 （这是唯一情况，可序列化事务块可以但可重复读事务不行。）另一方面，SIRead锁经常需要保持过去的事务提交，直到重叠读写事务完成。

可序列化事务一致性的使用可以简化开发。如果他们每次运行一个，保证任何一组并发序列化事务会具有相同的效果。 这意味着如果你能证明单一事务，作为书面的，当自己运行时将做正确事情， 你可以有信心它会在任何组合可序列化事务中做正确的事，即使没有任何有关那些其他事务的消息。 使用这种技术的环境中有一个处理序列化失败的方法是很重要的（它总是返回'40001'的SQLSTATE值）， 因为它很难准确预测，事务可能有助于读/写依赖并且需要回滚防止序列化异常。读/写依赖的监控是有成本的，正如序列化失败而终止之后进行事务重新启动， 但权衡成本和使用显式锁以及 `SELECT FOR UPDATE` 或者 `SELECT FOR SHARE` 涉及到的阻断， 可序列化事务在这种环境下是性能最好的选择。

为了最佳性能，当为并发控制依赖于可串行化事务时，应该考虑这些问题：

- 可能时作为 `只读` 声明事务。
- 如果需要，可以使用连接池，控制活动连接数。这是一个重要性能的考虑，但是在使用可串行化 事务的繁忙系统中尤其重要。

- 比起需要完整性目的来说不要将更多的东西放到单一事务中。
- 不要让连接在"闲置的事务"中停留超过需要的时间。
- 消除显示锁， `SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 不再需要，因为通过可串行化事务自动提供保护。
- 当系统强制连接多个页级别谓词锁到单一关系级别谓词锁，因为谓词锁表是短期存储，可能产生可串行化失败率的增加。你可以通过增加max_pred_locks_per_transaction 来避免。
- 顺序扫描总是需要一个关系级别谓词锁。这可能会导致序列化失败率增加。 这可能有助于鼓励减少random_page_cost 和/或增加cpu_tuple_cost的索引扫描使用。 一定要权衡任何事务回滚的减少，并且重新启动查询执行时间内的任何整体变化。

Warning

序列化事务隔离级别尚未被添加到热备复制目标中 （正如在Section 25.5中描述的）。 严格的隔离级别目前热备方式上支持可重复读。 当在主库上执行所有永久数据库写入可序列化事务中将确保所有的措施将最终达成一致， 运行在备库上的可重复读事务会看到一个过渡状态，与主库上的任何串行执行的可序列化事务不一致。

13.3. 明确锁定

PostgreSQL提供了多种锁模式用于控制对表中数据的并发访问。 这些模式可以用于在MVCC无法给出期望行为的场合。同样， 大多数PostgreSQL命令自动施加恰当的锁以保证被引用的表在命令的执行过程中不会以一种不兼容的方式被删除或者修改。 比如，在存在其它并发操作的时候， `TRUNCATE` 是不能在同一个表上面执行的

要检查数据库服务器里所有当前正在被持有的锁， 可以使用 `pg_locks` 系统视图。 有关监控锁管理器子系统状态的更多信息，请参考章Chapter 27。

13.3.1. 表级锁

下面的列表显示了可用的锁模式和它们被PostgreSQL自动使用的场合。 你也可以用LOCK命令明确获取这些锁。请注意所有这些锁模式都是表级锁， 即使它们的名字包含"row"单词(这些名称是历史遗产)。从某种角度而言， 这些名字反应了每种锁模式的典型用法— 但是语意却都是一样的。 两种锁模式之间真正的区别是它们有着不同的冲突锁集合(参见Table 13-2)。 两个事务在同一时刻不能在同一个表上持有相互冲突的锁。不过，一个事务决不会和自身冲突。比如，它可以在一个表上请求 `ACCESS EXCLUSIVE` 然后接着请求 `ACCESS SHARE` 。 非冲突锁模式可以被许多事务同时持有。请特别注意有些锁模式是自冲突的(比如，在任意时刻 `ACCESS EXCLUSIVE` 模式就不能够被多个事务拥有)， 但其它锁模式都不是自冲突的(比如， `ACCESS SHARE` 可以被多个事务持有)。

表级锁模式

`ACCESS SHARE`

只与 `ACCESS EXCLUSIVE` 冲突。

`SELECT` 命令在被引用的表上请求一个这种锁。通常， 任何只读取表而不修改它的命令都请求这种锁模式。

`ROW SHARE`

与 `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。

`SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 命令在目标表上需要一个这样模式的锁 (加上在所有被引用但没有 `ACCESS SHARE` 的表上的 `FOR UPDATE/FOR SHARE` 锁)。

`ROW EXCLUSIVE`

与 `SHARE` ， `SHARE ROW EXCLUSIVE` ， `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。

`UPDATE` ， `DELETE` 和 `INSERT` 命令自动请求这个锁模式(加上所有其它被引用的表上的 `ACCESS SHARE` 锁)。 通常，这种锁将被任何修改表中数据的查询请求。

`SHARE UPDATE EXCLUSIVE`

与 `SHARE UPDATE EXCLUSIVE` , `SHARE` , `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。 这个模式保护一个表不被并发模式改变和 `VACUUM` 。

`VACUUM` (不带 `FULL` 选项), `ANALYZE` , `CREATE INDEX CONCURRENTLY` 和 `ALTER TABLE` 请求这样的锁。

`SHARE`

与 `ROW EXCLUSIVE` , `SHARE UPDATE EXCLUSIVE` , `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。 这个模式避免表的并发数据修改。

`CREATE INDEX` (不带 `CONCURRENTLY` 选项)语句要求这样的锁模式。

`SHARE ROW EXCLUSIVE`

与 `ROW EXCLUSIVE` , `SHARE UPDATE EXCLUSIVE` , `SHARE` , `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。这个模式避免表的并发数据修改。 并且是自我排斥的，因此每次只有一个会话可以拥有它。

任何PostgreSQL命令都不会自动请求这个锁模式。

`EXCLUSIVE`

与 `ROW SHARE` , `ROW EXCLUSIVE` , `SHARE UPDATE EXCLUSIVE` , `SHARE` , `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` 和 `ACCESS EXCLUSIVE` 锁模式冲突。这个模式只允许并发 `ACCESS SHARE` 锁，也就是说， 只有对表的读动作可以和持有这个锁模式的事务并发执行。

任何PostgreSQL命令都不会在用户表上自动请求这个锁模式。

`ACCESS EXCLUSIVE`

与所有模式冲突(`ACCESS SHARE` , `ROW SHARE` , `ROW EXCLUSIVE` , `SHARE UPDATE EXCLUSIVE` , `SHARE` , `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` 和 `ACCESS EXCLUSIVE`)。这个模式保证其所有者(事务)是可以访问该表的唯一事务。

`ALTER TABLE` , `DROP TABLE` , `TRUNCATE` , `REINDEX` , `CLUSTER` 和 `VACUUM FULL` 命令要求这样的锁。 在 `LOCK TABLE` 命令没有明确声明需要的锁模式时，它是缺省锁模式。

> **Tip:** 只有 `ACCESS EXCLUSIVE` 阻塞 `SELECT` (不包含 `FOR UPDATE/SHARE` 语句)。

一旦请求已获得某种锁，那么该锁模式将持续到事务结束。但是如果在建立保存点之后才获得锁， 那么在回滚到这个保存点的时候将立即释放所有该保存点之后获得的锁。这与 `ROLLBACK` 取消所有保存点之后对表的影响的原则一致。 同样的原则也适用于PL/pgSQL异常块中获得的锁： 一个跳出块的错误将释放在块中获得的锁。

Table 13-2. 冲突锁模式

Requested Lock Mode	**Current Lock Mode**					
ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	E
---	---	---	---	---	---	-
ACCESS SHARE	X					
ROW SHARE	X	X				
ROW EXCLUSIVE	X	X	X	X		
SHARE UPDATE EXCLUSIVE	X	X	X	X	X	
SHARE	X	X	X	X	X	
SHARE ROW EXCLUSIVE	X	X	X	X	X	X
EXCLUSIVE	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X

13.3.2. 行级锁

除了表级锁以外，还有行级锁，他们可以是排他的或者是共享的。特定行上的排他行级锁是在行被更新的时候自动请求的。 该锁一直保持到事务提交或者回滚。行级锁不影响对数据的查询，它们只阻塞对同一行的写入。

要在不修改某行的前提下请求该行上的一个排他行级锁，用 `SELECT FOR UPDATE` 选取该行。 请注意一旦我们请求了特定的行级锁，那么该事务就可以多次对该行进行更新而不用担心冲突。

要在某行上请求一个共享的行级锁，用 `SELECT FOR SHARE` 选取该行。 一个共享锁并不阻止其它事务请求同一个共享的锁。不过，其它事务不允许更新、删除、 或者排他锁住持有共享锁的行。任何这么做的企图都将被阻塞并等待共享锁的释放。

PostgreSQL不会在内存里保存任何关于已修改行的信息，因此对一次锁定的行数没有限制。不过，锁住一行会导致一次磁盘写；因为 `SELECT FOR UPDATE` 将修改选中的行以标记它们被锁住了， 所以会导致磁盘写。

除了表级别和行级别的锁以外，页面级别的共享/排他锁也用于控制共享缓冲池中表页面的读/写。 这些锁在抓取或者更新一行后马上被释放。应用程序员通常不需要关心页级锁， 我们在这里提到它们只是为了完整。

13.3.3. 死锁

明确锁定的使用可能会增加死锁的可能性，死锁是指两个(或多个)事务相互持有对方期待的锁。 比如，如果事务 1 在表 A 上持有一个排他锁，同时试图请求一个在表 B 上的排他锁，而事务 2 已经持有表 B 的排他锁，而却正在请求在表 A 上的一个排他锁， 那么两个事务就都不能执行。 PostgreSQL能够自动侦测死锁条件并且会通过退出其中一个事务从而允许其它事务完成来解决这个问题。 具体哪个事务会被退出是很难预计的，而且也不应该依靠这样的预计。

要注意的是死锁也可能会因为行级锁而发生(即使是没有使用明确的锁定)。 考虑如下情况，两个并发事务在修改一个表。第一个事务执行了：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这样就在指定帐号的行上请求了一个行级锁。然后，第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一个 `UPDATE` 语句成功地在指定行上请求到了一个行级锁，因此它成功更新了该行。 但是第二个 `UPDATE` 语句发现它试图更新的行已经被锁住了，因此它等待持有该锁的事务结束。 事务二现在就在等待事务一结束，然后再继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一企图在指定行上请求一个行级锁，但是它得不到：事务二已经持有这样的锁了。 所以它等待事务二完成。因此，事务一被事务二阻塞住了，而事务二也被事务一阻塞住了： 这就是一个死锁条件。PostgreSQL将侦测这样的条件并退出其中一个事务。

防止死锁的最好方法通常是保证所有使用一个数据库的应用都以一致的顺序在多个对象上请求锁定。 在上面的例子里，如果两个事务以同样的顺序更新那些行，那么就不会发生死锁。 我们也要保证在一个对象上请求的第一个锁是该对象需要的最高的锁模式。 如果我们无法提前核实这些问题，那么我们可以通过在现场重新尝试因死锁而退出的事务的方法来处理。

只要没有检测到死锁条件，事务将一直等待表级锁或行级锁的释放。 这意味着一个事务持续的时间太长不是什么好事(比如等待用户输入)。

13.3.4. 咨询锁

PostgreSQL允许创建由应用定义其含义的锁。这种锁被称为咨询锁， 因为系统并不强迫其使用— 而是由应用来保证其被恰当的使用。 咨询锁可用于 MVCC 难以实现的锁定策略。 比如，咨询锁一般用于模拟常见于"平面文件"数据管理系统的悲观锁策略。 虽然可以用存储在表中的一个特定标志达到同样的目的，但是使用咨询锁更快，还可以避免表膨胀， 更可以在会话结束的时候由系统自动执行清理工作。

PostgreSQL中有两种方式可以获得咨询锁：会话级别或者事务级别。 咨询锁一旦被持有就将持续到被明确释放或会话结束。 不同于各种标准的锁，咨询锁并不考虑事务的语意： 在一个被回滚的事务中获得的咨询锁并不会被自动释放，同样的， 在一个失败的事务中释放的咨询锁仍将保持释放。同一个咨询锁可以被它自己的进程多次获得： 对于每一个锁定请求必须有一个相应的释放请求，这样才能最终真正释放该锁。 另一方面，事务级别的锁请求，表现得更像普通锁请求：他们结束事务时自动释放， 并且没有明确的解锁操作。这种行为通常比咨询锁的短期使用会话级别行为更方便。 会话级别和事务级别锁请求为相同的咨询锁标识符将以预期方式互相阻止。 如果某个会话已经持有一个咨询锁，那么对该锁的多次锁定请求将总会成功，即使其它会话正在等候该锁的释放也是如此。 不管是否持有已存在锁，并且新的要求是会话级别或者事务级别，这个语句是真的。

与PostgreSQL中其它锁一样， 可以在 `pg_locks` 系统视图中查看当前被会话持有的所有咨询锁。

咨询锁和规则锁存储在共享内存池中，其中大小由max_locks_per_transaction和max_connections配置参数决定。 千万不要耗尽这些内存，否则服务器将不能再获取任何新锁。 因此服务器可以获得的咨询锁数量是有限的，根据服务器的配置不同， 这个限制可能是几万到几十万个。

在某些使用咨询锁方法的特定情况下，特别是查询包括明确的排序或 `LIMIT` 子句的时候， 由于 SQL 表达式求值顺序的影响，必须注意控制咨询锁的获取。例如：

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

在上述查询中，第二种形式是危险的，因为 `LIMIT` 并不一定在锁定函数执行之前被应用。 这可能导致获得某些应用不期望的锁，并因此在会话结束之前无法释放。从应用的角度来看，这样的锁将被挂起，虽然它们仍然在 `pg_locks` 中可见。

用于操作咨询锁的函数在Section 9.26.8中描述。

13.4. 应用层数据完整性检查

关于使用读已提交事务的数据完整性强制执行业务规则是很难的，由于数据视图从每个语句偏移，如果发生写入冲突，则单一语句可以不限制自身到语句快照。

当可重复读事务在执行期间有稳定的数据表视图， 使用MVCC数据一致性检查的快照有一个微妙的问题，涉及一些被称为读/写冲突的东西。 如果一个事务写入数据，并且并发事务尝试读取同一数据（无论写之前还是之后）， 它不能看到其他事务的工作。读者似乎首先被执行，无论哪个先被启动或先被提交。 如果是这样，这是没有问题的，但如果读者也写由并发事务读取的数据， 则现在有一个前面已提到的事务似乎运行着的事务。 如果已经被执行的事务最后首先被提交，出现在事务执行顺序图中，这样一个循环是很容易的。 当这样一个循环出现时，完整性检查将不能正常工作，而没有一定的帮助。

正如Section 13.2.3提及到的，可串行化事务是可重复读事务，其中添加了 读/写冲突危险模式的非阻塞监测。当这个模式被监测时，有可能导致明显的执行顺序周期，其中所 涉及到的事务回滚以打破这个周期。

13.4.1. 可串行化事务执行一致性

如果序列化事务隔离级别是用于所有写和所有读，这需要数据一致视图，没有其他努力来确保一致性。 来自其他环境的软件书面使用序列化事务确保一致性应该在PostgreSQL这方面"只是工作"。

当使用这种技术时，如果应用软件完成框架，其中回滚到 可串行化失败的地方进行自动重启事务，它可以避免为应用参数产生不必要的负担。 设置 `default_transaction_isolation` 到 `可串行化` 是一个很好的主意。 明智的采取一些措施确保没有其他可用的事务隔离级别，或者是无意的或者破坏完整性检查， 通过触发器中的事务隔离级别检查。

参见Section 13.2.3获取性能建议。

> **Warning**
>
> 这一级别的完整性保护使用序列化事务还没有延伸到热备模式（Section 25.5）。 因此，那些采用热备方式可能要使用可重复读，并且主库上明确锁。

13.4.2. 明确阻塞锁的执行一致性

当可能进行非串行化写入时，要保证一行当前实际存在和避免其被同时更新， 我们必须使用 `SELECT FOR UPDATE` ，`SELECT FOR SHARE` 或者合适的 `LOCK TABLE` 语句。 `SELECT FOR UPDATE` 和 `SELECT FOR SHARE` 只是对其它的并发更新锁住返回的行，

而 `LOCK TABLE` 保护整个表。当从其它环境向PostgreSQL里用可串行化模式移植应用时一定要把这些问题考虑进去。

还要注意来自其他环境的这些转变事实： `SELECT FOR UPDATE` 不能保证并发事务不更新或删除已选择的行。 为了这样做，PostgreSQL中你必须更新行，即使没有值需要被改变。 来自获取同一个锁或者执行 `UPDATE` 或者 `DELETE` 的 `SELECT FOR UPDATE` 暂时阻塞 其他事务可能影响锁定行，但是一旦事务持有这个锁提交或者回滚，被阻塞事务将继续做冲突操作，除非 当锁被持有的时候，执行实际 `UPDATE` 行。

在MVCC非串行化下，全局有效性检查需要一些额外的考虑。 比如，一个银行应用可能会希望检查一个表中的所有扣款总和等于另外一个表中的加款总和， 同时两个表还会被活跃地更新。在读已提交模式下比较两个连续的 `SELECT sum(...)` 命令的结果是不可靠的， 因为第二个查询很可能会包含第一个没计算的事务提交的结果。 在一个可串行化的事务里进行两个求和则给出在可串行化事务开始之前提交的所有事务产生的精确的结果— 但我们还是会合理地置疑在结果提交的时候，它们是否还相关。 如果可串行化事务本身在试图做一致性检查之前进行了某些变更，那么检查的有用性就更加值得讨论了， 因为现在它包含了一些(但不是全部)事务开始后的变化。在这种情况下， 一个仔细的人会希望锁住所有需要检查的表，这样才能获得一个无可置疑的当前现状的图像。 一个 `SHARE` 模式(或者更高级)的锁保证在被锁定表中除了当前事务之外，没有未提交的更新。

还要注意如果我们依赖明确锁定来避免并发更新，那么我们应该使用读已提交模式， 或者是在可串行化模式里在执行命令之前小心地获取锁。 在可串行化事务里获取的锁保证了不会有其它正在运行的修改该表的事务存在， 但是如果事务看到的快照提前获取了锁，那么它可能提前把一些现在已经提交的改变放到表中。 一个可串行化事务的快照实际上是在它的第一个查询或者数据修改命令(`SELECT` , `INSERT` , `UPDATE` , or `DELETE`)开始的时候冻结的， 因此我们可以在快照冻结之前明确获取锁。

13.5. 锁和索引

尽管PostgreSQL提供对表数据访问的非阻塞的读/写， 但并非所有PostgreSQL里实现的索引访问模式都能够进行非阻塞读/写。 不同的索引类型按照下面方法操作：

B-tree, GiST and SP-GiST indexes

短期的页面级共享/排他锁用于读/写访问。锁在索引行被插入/抓取后立即释放。 这种索引类型提供了无死锁条件的最高级的并发性。

Hash indexes

Hash 桶级别的共享/排他锁用于读/写访问。锁在整个 Hash 桶处理完成后释放。 Hash 桶级锁比索引级的锁提供了更好的并发性但是可能产生死锁， 因为锁持有的时间比一次索引操作时间长。

GIN indexes

短期的页面级共享/排他锁用于读/写访问。锁在索引行被插入/抓取后立即释放。 但要注意的是一个GIN索引值的插入通常导致几个每行几个索引键的插入， 因此GIN可能为了插入一个值而做大量的工作。

目前，B-tree 索引为并发应用提供了最好的性能。因为它还有比 Hash 索引更多的特性，在那些需要对标量数据进行索引的并发应用中，我们建议使用 B-tree 索引类型。在处理非标量类型数据的时候，B-tree 就没什么用了，应该使用 GiST 或 GIN 索引。

Chapter 14. 性能提升技巧

Table of Contents

查询的性能可能受多种因素影响。其中一些因素可以由用户操纵， 而其它的则属于下层系统设计的基本问题了。 本章我们提供一些有关理解和调节PostgreSQL性能的线索。

14.1. 使用 `EXPLAIN`

PostgreSQL对每个查询产生一个查询规划。 为匹配查询结构和数据属性选择正确的规划对性能绝对有关键性的影响。 因此系统包含了一个复杂的规划器用于寻找最优的规划。 你可以使用EXPLAIN命令察看规划器为每个查询生成的查询规划是什么。 阅读查询规划是一门值得专门写一厚本教程的学问，但是这部分试图掩盖这些基本信息。

本节的例子是从数据库执行 `VACUUM ANALYZE` 之后的回归测试中提取的，使用9.3开发源。 如果你尝试自己的例子，你应该可以得到类似结果，但你的估计成本及行数可能会略有不同，因为 `ANALYZE` 的统计数据是随机样本，而不是确切的，并且因为成本本身有点依赖于平台。

该示例使用 `EXPLAIN` 的缺省"文本"输出格式， 它结构紧凑，便于人们阅读。如果你想为进一步分析提供 `EXPLAIN` 输出给程序， 你应该使用它的机器可读的输出格式之一(XML, JSON, or YAML)来代替。

14.1.1. `EXPLAIN` 基础

查询规划的结构是一个规划节点的树。最底层的节点是表扫描节点： 它们从表中返回原始数据行。不同的表访问模式有不同的扫描节点类型： 顺序扫描、索引扫描、位图索引扫描。 也有非表行来源，如 `VALUES` 子句和 `FROM` 中的设置返回函数， 其中有他们自己的扫描节点类型。 如果查询需要连接、聚集、排序、或者对原始行的其它操作， 那么就会在扫描节点之上有其它额外的节点。并且，做这些操作通常都有多种方法， 因此在这些位置也有可能出现不同的节点类型。 `EXPLAIN` 给规划树中每个节点都输出一行， 显示基本的节点类型和规划器为执行这个规划节点预计的开销值。 其他行可能会出现，从节点的汇总行缩进，以显示节点的附加属性。 第一行(最上层的汇总行节点)是对该规划的总执行开销的预计；这个数值就是规划器试图最小化的数值

这里是一个简单的例子，只是用来显示输出会有些什么内容：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于此查询没有 `WHERE` 子句，它必须扫描所有表的行，所以规划器已经选择使用一个简单的顺序扫描计划。 括号中引用的数值是（从左到右）：

- 预计的启动开销。在输出扫描开始之前消耗的时间，也就是在一个排序节点里执行排序的时间。

- 预计总开销。这是假设所规定的，计划节点运行完成，即所有可用行被检索。 在实践中一个节点的父节点可能会很快停止读取所有可用的行（参见 `LIMIT` 下面的例子）。
- 预计这个规划节点输出的行数。同样，只执行到完成为止。
- 预计这个规划节点的行平均宽度(以字节计算)。

开销是用规划器根据成本参数(参见节Section 18.7.2)捏造的单位来衡量的， 习惯上以磁盘页面抓取为单位。 也就是seq_page_cost将被按照习惯设为 `1.0` (一次顺序的磁盘页面抓取)，其它开销参数将参照它来设置。本节的例子都假定这些参数使用默认值。

有一点很重要：一个上层节点的开销包括它的所有子节点的开销。还有一点也很重要： 这个开销只反映规划器关心的东西，尤其是没有把结果行传递给客户端的时间考虑进去， 这个时间可能在实际的总时间里占据相当重要的分量，但是被规划器忽略了， 因为它无法通过修改规划来改变：我们相信，每个正确的规划都将输出同样的记录集。

`行` 值有一些小技巧，因为它不是规划节点处理/扫描过的行数， 而是节点发射数目。通常会少于扫描数，正如应用于此节点上的任意 `WHERE` 子句条件的过滤结果。通常而言， 顶层的行预计会接近于查询实际返回、更新、删除的行数。

回到我们的例子：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的获得非常直截了当。如果你这样做：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

你会发现 `tenk1` 有 358 磁盘页面和 10000 行。 估计成本作为（磁盘页面读取 *seq_page_cost*）+（行扫描cpu_tuple_cost）被计算。默认情况下， `seq_page_cost` 是 1.0， `cpu_tuple_cost` 是0.01, 因此估计成本为(358 *1.0) + (10000* 0.01) = 458。

现在让我们修改查询并增加一个 `WHERE` 条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

请注意 `EXPLAIN` 输出显示 `WHERE` 子句当作一个"filter"条件附属于顺序扫描计划节点。 这意味着规划节点为它扫描的每一行检查该条件，并且只输出符合条件的行。 预计的输出行数降低了，因为有 `WHERE` 子句。不过，扫描仍将必须访问所有 10000 行， 因此开销没有降低；实际

上它还增加了一些（确切的说，通过10000 * cpu_operator_cost）以反映检查 `WHERE` 条件的额外CPU时间。

这条查询实际选择的行数是7000，但是预计的 `行数` 只是个大概。如果你试图重复这个试验，那么你很可能得到不同的预计。还有，这个预计会在每次 `ANALYZE` 命令之后改变， 因为 `ANALYZE` 生成的统计是从该表中随机抽取的样本计算的。

把查询限制条件改得更严格一些：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

这里，规划器决定使用两步的规划：最底层的规划节点访问一个索引，找出匹配索引条件的行的位置， 然后上层规划节点真实地从表中抓取出那些行。独立地抓取数据行比顺序地读取它们的开销高很多， 但是因为并非所有表的页面都被访问了，这么做实际上仍然比一次顺序扫描开销要少。 使用两层规划的原因是因为上层规划节点把索引标识出来的行位置在读取它们之前按照物理位置排序， 这样可以最小化独立抓取的开销。节点名称里面提到的"bitmap"是进行排序的机制。

现在让我们添加另外一个条件到 `WHERE` 子句：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

新增的条件 `stringu1 = 'xxx'` 减少了预计的输出行，但是没有减少开销， 因为我们仍然需要访问相同的行。 请注意， `stringu1` 子句不能当做一个索引条件使用(因为这个索引只是在 `unique1` 列上有)。 它被当做一个从索引中检索出的行的过滤器来使用。 因此开销实际上略微增加了一些以反映这个额外的检查。

在某些情况下规划区更加喜欢"simple"索引扫描规划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

在这种规划类型中，表的数据行是以索引顺序抓取的，这样就令读取它们的开销更大， 但是这里的行少得可怜，因此对行位置的额外排序并不值得。最常见的就是看到这种规划类型只抓取一行， 以及那些要求 `ORDER BY` 条件匹配索引顺序的查询。因为那时候没有多余的排序步骤是必要的以满足 `ORDER BY` 。

如果在 `WHERE` 里面使用的好几个字段上都有索引，那么规划器可能会使用索引的AND或OR的组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

但是这么做要求访问两个索引，因此与只使用一个索引，而把另外一个条件只当作过滤器相比， 这个方法未必是更优。如果你改变涉及的范围，你会看到规划器相应地发生变化。

下面是一个例子，显示 `LIMIT` 的影响：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

这是上面相同的查询，但我们增加了 `LIMIT` ，以致于不是所有的行需要被检索，并且规划关于该怎么做改变了主意， 请注意，索引扫描节点的总成本和行数被显示，好像它是运行完毕的。然而，限制节点预计在提取这些行的仅仅五分之一后停止， 所以其总成本只有五分之一之多，这就是实际的预算费用查询。该计划优于增加一个限制节点到 先前的计划，因为该限制无法避免支付位图扫描的启动成本，所以总成本将超过使用这种方法的25个单位的东西。

让我们试着使用我们上面讨论的字段连接两个表：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个规划中，我们有两个表扫描的作为输入或者子节点的嵌套循环连接节点， 节点摘要行的缩进反映规划树结构。 连接的第一个，或者"outer"，子节点就是类似于我们之前看到的位图扫描。 其成本和行数是一样的，正如我们从 `SELECT ... WHERE unique1 < 10` 获得。 因为我们只能在那个节点上应用 `WHERE` clause `unique1 < 10` 。 `t1.unique2 = t2.unique2` 子句还没有任何关系。因此它不影响外层扫描的行计数。 嵌套循环连接节点将运行它的第二部分， 或者"inner"子节点一次从外部子节点获得每一行。 从目前的外层行获得的值可以被插入到内扫描。这儿，从外层行中获得的 `t1.unique2` 是可用的。 这样我们就得到一个计划和成本，并且类似于我们上面看到的简单的 `SELECT ... WHERE t2.unique2 =` `_constant_` 的情况。（估计费用实际上比上面看到的低一点，在 `t2` 上可重复的索引扫描期间，作为期望发生的高速缓存结果）。 以外层扫描的开销为基础设置循环节点的开销，加上每个外层行的一个重复(这里是10 * 7.87)， 然后再加上连接处理需要的一点点CPU时间。

在这个例子里，连接的输出行数与两个扫描的行数的乘积相同，但通常并不是这样的， 因为通常你会有提及两个表的 `WHERE` 子句，因此它只能应用于连接(join)点， 而不能影响两个关系的输入扫描。 这里有一个例子：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                         QUERY PLAN
---------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=2
               Index Cond: (unique2 < 10)
```

条件 `t1.hundred < t2.hundred` 不能 在 `tenk2_unique2` 索引中被测试，因此它被应用在 连接节点。这减少了连接节点的预计输出行数， 但不改变任何一个输入扫描。

注意，这里的规划器已经选择"具体化"连接内部关系， 通过放在规划节点上面。这也就是说， `t2` 索引扫描将执行一次，尽管 嵌套循环连接节点需要读取数据十次，来自外部关系的每一行。 实现节点将数据保存在存储器中，因为它被读取，然后从存储器每个后续过程中返

回每一个数据。

当与外部联接时，你可能会看到带有附属"Join Filter"以及纯"Filter"条件的连接计划节点。 连接过滤条件来自于外部连接的 `ON` 子句，因此 这样的行失败了，连接过滤条件仍然可以作为非扩展行发出。 但一个纯过滤条件可以在外连接规则之后被应用 因此这个行为无条件地删除行。在内连接中 这些过滤器类型之间没有语义差异。

如果我们改变查询的选择性，我们可能会得到一个非常不同的连接计划:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

在这里，规划器选择使用一个哈希联接，表中的行被输入到内存中的哈希表中，在此之后，其他 表被扫描并且哈希表进行探测以匹配每一行。 再次注意如何缩进来反映规划结构：在 `tenk1` 上的位图 扫描是输入到哈希节点，它构造哈希表。这之后返回哈希连接节点，其内容是从它的外部子计划中读取每一行并且搜索每一个哈希表。

另一种可能的连接类型是合并连接，在这里说明：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

合并连接要求其输入的数据在连接键上进行排序。在这种 规划中 `tenk1` 数据是通过使用索引扫描访问正确顺序的行来进行排序。 但顺序扫描和排序是 `onek` 的首选，因为有该表上被访问的更多行。 （顺序扫描和排序为排序行数而频繁进行索引扫描，因为通过索引扫描需要不连续的磁盘访问）

找另外一个规划的方法是通过设置每种规划类型的允许/禁止开关(在Section 18.7.1里描述)，强制规划器抛弃它认为优秀的(扫描)策略。这个工具目前比较原始，但很有用。又见Section 14.3。 例如，如果我们不相信顺序扫描和排序对于前面例子中处理表 `onek` 是最好的方式，我们可以尝试

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

这表明规划期认为通过索引扫描排序 `onek` 比顺序扫描和排序更昂贵约12%。 当然，接下来的问题是它是否是对的。我们可以使用 `EXPLAIN ANALYZE` 调查，正如下面所讨论的：

14.1.2. `EXPLAIN ANALYZE`

我们可以用 `EXPLAIN` 的 `ANALYZE` 检查规划器的估计值的准确性。 这个命令实际上执行该查询然后显示每个规划节点内实际运行时间的和以及单纯 `EXPLAIN` 显示的估计开销。 比如，我们可以像下面这样获取一个结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                           QUERY PLAN
------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10 loo
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual time=0.
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0) (actua
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244) (ac
         Index Cond: (unique2 = t1.unique2)
 Total runtime: 0.501 ms
```

请注意"actual time"数值是以真实时间的毫秒计的，而 `cost` 估计值是以任意磁盘抓取的单元计的； 因此它们很可能不一致。我们要关心的事是两组比值是否一致。 通常最重要的事情是看是否估计行数相当接近于现实。在这个例子中， 估计都是完全正确的，但是这是相当不寻常的做法。

在一些查询规划里，一个子规划节点很可能运行多次。比如，在上面的嵌套循环的规划里，内层的索引扫描对每个外层行执行一次。在这种情况下， `loops` 报告该节点执行的总数目，而显示的实际时间和行数目是每次执行的平均值。这么做的原因是令这些数字与开销预计显示的数字具有可比性。 要乘以 `loops` 值才能获得在该节点花费的总时间。在上面的例子中，我们共需要0.220毫秒来执行 `tenk2` 的索引扫描。

在某些情况下 `EXPLAIN ANALYZE` 显示超出规划节点执行时间和行数的额外执行统计数据。 例如，排序和哈希节点提供额外的信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                 QUERY PLAN
------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 row
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 wi
                           Index Cond: (unique1 < 100)
 Total runtime: 8.008 ms
```

排序节点显示使用的排序方法（特别是，排序是否在内存或磁盘上）以及所需的内存或磁盘空间量。 哈希节点显示哈希桶数量以及批处理用于哈希表的内存峰值数。 （如果批处理数大于1，也将有参与磁盘空间使用情况，但不在这显示。

另一种类型的附加信息是通过过滤条件删除行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                     QUERY PLAN
------------------------------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 row
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Total runtime: 5.905 ms
```

这些计数对于应用在连接节点的过滤条件特别有价值。 "删除行"只出现在扫描行，或者连接节点的情况下的潜在连接对， 通过过滤条件被拒绝。

类似的情况，过滤条件产生"lossy"的索引扫描。 例如，考虑多边形这个搜索包含的具体点：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                              QUERY PLAN
------------------------------------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 row
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Total runtime: 0.083 ms
```

规划器认为（很正确）这种表太小而干扰索引扫描，所以我们有一个纯顺序扫描， 其中所有行通过过滤条件被拒绝。但是，如果我们强制 使用索引扫描，我们看到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual t
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Total runtime: 0.144 ms
```

在这里，我们可以看到，索引返回一个候选行，这是 通过索引条件的重新检查被拒绝。这是因为 GiST索引对于多边形封闭测试是"lossy"：它实际上 返回带有重叠目标多边形的行，然后我们在那些行上进行确切的封闭测试。

`EXPLAIN` 有 `BUFFERS` 选项， `ANALYZE` 以获得更多的运行时间统计：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                           QUERY PLAN
------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.3
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 lo
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actu
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (act
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Total runtime: 0.423 ms
```

通过 `BUFFERS` 提供的数字帮助辨识查询的哪些部分大多是I/O密集型。

请记住，因为 `EXPLAIN ANALYZE` 实际运行查询， 任何副作用还是一样会发生，即使无论什么结果查询可能的输出都将被丢弃而 赞成输出 `EXPLAIN` 的数据。 如果要分析一个数据修改的查询，而无需改变你的表，你可以回滚命令到后面，例如：

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                           QUERY PLAN
--------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=0.1
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actu
               Index Cond: (unique1 < 100)
 Total runtime: 14.727 ms

ROLLBACK;
```

如该示例中，当查询是 `INSERT` , `UPDATE` 或者 `DELETE` 命令时， 申请表变化的实际工作是由顶层插入、更新、或删除规划节点完成的。 这个节点下的规划节点进行定位旧的行和/或计算新数据。 所以上面，我们看到了已经看到的相同排序的位表扫描， 并且其输出被传递给存储更新行的更新节点。 值得一提的是，虽然修改数据的节点可以采取大量的运行时间（在这里，它消耗了大部分的共享的时间）， 规划器目前不添加任何东西的成本来估计说明这项工作。 这是因为要做的工作是同样为了每一个正确的查询规划， 因此它不影响规划决定。

`EXPLAIN ANALYZE` 显示的 `Total runtime` 包括执行器启动和关闭的时间， 以及被激发的任何触发器运行时间。但它不包括分析、重写、规划的时间。 执行 `BEFORE` 触发器花费的时间，如果有的话，包括在为相关插入，更新或删除节点的时间内， 但执行 `AFTER` 触发器的时间花费并不计算在内， 因为整个规划完成之后，才触发 `AFTER` 触发器。 单独显示每个触发器花费的总时间（ `BEFORE` 或者 `AFTER` ）。 需要注意的是延迟约束触发器直到事务结束将不会执行，因而不会通过 `EXPLAIN ANALYZE` 显示。

14.1.3. 警告

有两个显著方式测量运行时间，通过 `EXPLAIN ANALYZE` 偏离相同查询的正常执行。 首先，由于没有输出行被传递到客户端， 不包含网络传输成本和I/O转换费用。其次，通过 `EXPLAIN ANALYZE` 增加的测量开销是巨大的， 特别是在慢的 `gettimeofday()` 操作系统调用机器上。您可以使用 pg_test_timing工具来测量您系统上的定时开销。

`EXPLAIN` 的结果除了在你实际测试的情况之外不能推导出其它的情况； 比如，在一个小得像玩具的表上的结果不能适用于大表。规划器的开销计算不是线性的， 因此它很可能对大些或者小些的表选择不同的规划。一个极端的例子是一个只占据一个磁盘页面的表， 在这样的表上，不管它有没有索引可以使用，你几乎都总是得到顺序扫描规划。 规划器知道不管在任何情况下它都要进行一个磁盘页面的读取， 所以再扩大几个磁盘页面读取以查找索引是没有意义的。 （我们可以从 `polygon_tbl` 上面的例子中看到）

在某些情况中，实际与估计值不能很好的匹配，但没有什么是真的错了。 出现这样的一个情况，当规划节点执行是由 `LIMIT` 或类似效果而短暂停止 。例如，我们以前使用 `LIMIT` 查询。

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                          QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244) (act
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Total runtime: 0.336 ms
```

为索引扫描节点的估计成本和行数都被显示。即使它是运行完毕的。但现实是请求行运行两个之后限制节点停止， 所以实际的行数只有2和，并且运行时间低于成本估算。这不是估计错误，由于只有一个差异估计和真实值显示。

合并连接也有可混淆粗心的测量产品。 合并连接将停止读取一个输入，如果它用尽了其他输入，并且输入端的下一个关键值大于其他输入的最后一个关键值 ； 在这种情况下，就不可能有更多的匹配，所以不需要扫描第一个输入的其余部分。 这会导致无法读取所有子节点，有些像那些提到的 `LIMIT` 结果。 此外，如果外部（第一）子节点包含重复键值的行，内部（第二个）子节点被备份并重新扫描行匹配键值的部分。 `EXPLAIN ANALYZE` 计算同一内部行的重复部分，好像他们是真正的附加行。 当有许多外部重复部分，为了内部子规划节点比真实内部关系行数足够大，则报告的实际行数。

BitmapAnd和BitmapOr节点总是报告自己的实际行数为零，由于实施限制。

14.2. 规划器使用的统计信息

就像我们在上一节里展示的那样，查询规划器需要估计一个查询检索的行数， 这样才能选择正确的查询规划。本节就系统用于这些估计的统计进行一些描述。

统计的一个部分就是每个表和索引中的记录总数，以及每个表和索引占据的磁盘块数。 这个信息保存在 pg_class 表的 reltuples 和 relpages 字段中。 我们可以用类似下面的查询检索这些信息：

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

我们在这里可以看到 tenk1 有10000行，它的索引也有这么多行，但是索引远比表小得多(很正常)。

出于效率考虑， reltuples 和 relpages 不是实时更新的， 因此它们通常包含可能有些过时的数值。它们被 VACUUM ， ANALYZE 和几个DDL命令 (比如 CREATE INDEX)更新。 VACUUM 或者 ANALYZE 操作不扫描整个表（这是常见的情况）， 将逐步基于扫描表的部分更新 reltuples 数，从而产生一个近似值。 在任何情况下，规划器将把 pg_class 表里面的数值调整为和当前的物理表尺寸匹配， 以此获取一个更接近的近似值。

大多数查询只是检索表中行的一部分，因为它们有限制待查行的 WHERE 子句。 因此规划器需要对 WHERE 子句的选择性进行评估， 选择性也就是符合 WHERE 子句中每个条件的部分。 用于这个目的的信息存储在 pg_statistic 系统表中。 在 pg_statistic 中的记录是由 ANALYZE 和 VACUUM ANALYZE 命令更新的， 并且总是近似值，即使刚刚更新完也不例外。

除了直接查看 pg_statistic 之外， 我们手工检查统计的时候最好查看 pg_stats 的视图。 pg_stats 被设计成更容易可读的。 而且， pg_stats 是所有人都可以读取的， 而 pg_statistic 只能由超级用户读取。 这样就可以避免非特权用户从统计信息中获取一些和其他人的表内容相关的信息。 pg_stats 视图是受约束的， 只显示当前用户可读的表。比如，我们可以：

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp
 name    | t         |  -0.284859 | I- 880                        Ramp+
         |           |            | I- 580                        Ramp+
         |           |            | I- 680                        Ramp+
         |           |            | I- 580                            +
         |           |            | State Hwy 13                  Ramp
(2 rows)
```

需要注意的是两行显示为同一列，1对应在 `road` 表(`inherited` = `t`)开头的完整继承层次结构。 而另一个只包含 `road` 表本身(`inherited` = `f`)。

在 `pg_statistic` 中通过 `ANALYZE` 存储的信息的数量， 特别是给每个字段用的 `most_common_vals` 和 `histogram_bounds` 数组上的最大记录数目 可以用 `ALTER TABLE SET STATISTICS` 命令设置， 或者是用运行时参数default_statistics_target进行全局设置。 目前缺省的限制是 100个记录。 提升该限制应该可以做出更准确的规划器估计，特别是对那些有不规则数据分布的字段而言， 代价是在 `pg_statistic` 里使用了更多空间，并且需要略微多一些的时间计算估计数值。相比之下， 比较低的限制可能更适合那些数据分布比较简单的字段。

有关规划器使用统计信息的进一步详情可参阅Chapter 60。

14.3. 用明确的 `JOIN` 控制规划器

我们可以在一定程度上用明确的 `JOIN` 语法控制查询规划器。要明白为什么有这茬事， 我们首先需要一些背景知识。

在简单的连接查询里，比如:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以按照任何顺序自由地连接给出的表。比如， 它可以生成一个查询规划先用 `WHERE` 条件 `a.id = b.id` 把 A 连接到 B， 然后用另外一个 `WHERE` 条件把 C 连接到这个表上来， 或者也可以先连接 B 和 C 然后再连接 A，同样得到这个结果。 或者也可以连接 A 到 C 然后把结果与 B 连接—不过这么做效率比较差， 因为必须生成完整的 A 和 C 的迪卡尔积，而在查询里没有可用的 `WHERE` 子句可以优化该连接(PostgreSQL执行器里的所有连接都发生在两个输入表之间， 所以在这种情况下它必须先得出一个结果)。重要的一点是这些连接方式给出语义上相同的结果， 但在执行开销上却可能有巨大的差别。因此，规划器会对它们进行检查并找出最高效的查询规划。

如果查询只涉及两或三个表，那么在查询里不会有太多需要考虑的连接。 但是潜在的连接顺序的数目随着表数目的增加程指数增加的趋势。 当超过十个左右的表以后，实际上根本不可能对所有可能做一次穷举搜索， 甚至对六七个表都需要相当长的时间进行规划。如果有太多输入的表， PostgreSQL规划器将从穷举搜索切换为基因概率搜索，以减少可能性数目(样本空间)。 切换的阈值是用运行时参数geqo_threshold设置的。基因搜索花的时间少， 但是并不一定能找到最好的规划。

当查询涉及外部连接时，规划器就不像对付普通(内部)连接那么自由了。比如，看看下面这个查询:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询的约束和前面一个非常相似，但它们的语义却不同， 因为如果 A 里有任何一行不能匹配B和C的连接里的行，那么该行都必须输出。因此这里规划器对连接顺序没有什么选择： 它必须先连接 B 到 C， 然后把 A 连接到该结果上。因此， 这个查询比前面一个花在规划上的时间少。 在其它情况下，规划器就有可能确定多种连接顺序都是安全的。比如， 对于:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有 `FULL JOIN` 完全强制连接顺序。 大多数 `LEFT JOIN` 或者 `RIGHT JOIN` 都可以在某种程度上重新排列。

明确的连接语法(`INNER JOIN` , `CROSS JOIN` 或无修饰的 `JOIN`)语义上和 `FROM` 中列出输入关系是一样的， 因此我们没有必要约束连接顺序。

即使大多数 `JOIN` 并不完全强迫连接顺序，但仍然可以明确的告诉PostgreSQL 查询规划器 `JOIN` 子句的连接顺序。 比如，下面三个查询逻辑上是等效的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果我们告诉规划器遵循 `JOIN` 的顺序，那么第二个和第三个还是要比第一个花在规划上的时间少。 这个作用对于只有三个表的连接而言是微不足道的，但对于数目众多的表，可能就是救命稻草了。

要强制规划器遵循准确的 `JOIN` 连接顺序，我们可以把运行时参数join_collapse_limit设置为1(其它可能的数值在下面讨论)。

你完全不必为了缩短搜索时间来约束连接顺序，因为在一个简单的 `FROM` 列表里使用 `JOIN` 操作符就很好了。 比如考虑：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置 `join_collapse_limit` = 1，那么这句话就相当于强迫规划器先把A连接到B， 然后再连接到其它的表上，但并不约束其它的选择。在本例中，可能的连接顺序的数目减少了 5 倍。

按照上面的想法考虑规划器的搜索问题是一个很有用的技巧， 不管是对减少规划时间还是对引导规划器生成好的规划都很有帮助。 如果缺省时规划器选择了一个糟糕的连接顺序，你可以用 `JOIN` 语法强迫它选择一个更好的— (假设知道一个更好的顺序)。所以我们建议多试验。

一个非常相近的影响规划时间的问题是把子查询压缩到它们的父查询里面。比如，考虑下面的查询:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这个情况可能在那种包含连接的视图中出现；该视图的 `SELECT` 规则将被插入到引用视图的场合， 生成非常类似上面的查询。通常，规划器会试图把子查询压缩到父查询里，生成:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这样通常会生成一个比独立的子查询更好些的规划。比如， 外层的 `WHERE` 条件可能先把X连接到 A 上，这样就消除了 A 中的许多行， 因此避免了形成全部子查询逻辑输出的需要。但是同时，我们增加了规划的时间 ；在这里，我们有一个用五路连接代替两个独立的三路连接的问题，这样的差距是巨大的， 因为可能的规划数的是按照指数增长的。规划器将在父查询可能超过 `from_collapse_limit` 个 `FROM` 项的时候， 不再压缩子查询，以此来避免巨大的连接搜索数。 你可以通过调整这个运行时参数来在规划时间和规划质量之间作出平衡。

from_collapse_limit和join_collapse_limit 名字类似是因为他们做的事情几乎相同：一个控制规划器何时把子查询"平面化"， 另外一个控制何时把明确的连接平面化。通常， 你要么把 `join_collapse_limit` 设置成和 `from_collapse_limit` 一样(明确连接和子查询的行为类似)，要么把 `join_collapse_limit` 设置为 1(如果你想用明确连接控制连接顺序)。 但是你可以把它们设置成不同的值，这样你就可以在规划时间和运行时间之间进行仔细的调节。

14.4. 向数据库中添加记录

第一次填充数据库时可能需要做大量的表插入。下面是一些建议，可以尽可能高效地处理这些事情。

14.4.1. 关闭自动提交

当使用多条 `INSERT` 时，关闭自动提交，并且只在每次(数据拷贝)结束的时候做一次提交。 在纯SQL里，这就意味着在开始的时候发出 `BEGIN` 并且在结束的时候执行 `COMMIT` 。 有些客户端的库可能背着你干这些事情， 这种情况下你必须确信只有在你确实要那些库干这些事情的时候它才做。 如果你允许每个插入都独立地提交，那么PostgreSQL会为所增加的每行记录做大量的处理。 在一个事务里完成所有插入的动作的最大的好处就是，如果有一条记录插入失败， 那么，到该点为止的所有已插入记录都将被回滚， 这样你就不会很难受地面对一个只装载了一部分数据的表。

14.4.2. 使用 `COPY`

使用COPY在一条命令里装载所有记录， 而不是一连串的 `INSERT` 命令。 `COPY` 命令是为装载数量巨大的数据行优化过的； 它没 `INSERT` 那么灵活，但是在大量装载数据的情况下，导致的荷载也少很多。 因为 `COPY` 是单条命令，因此填充表的时候就没有必要关闭自动提交了。

如果你不能使用 `COPY` ，那么使用PREPARE来创建一个预备 `INSERT` ， 然后使用 `EXECUTE` 多次效率更高。这样就避免了重复分析和规划 `INSERT` 的开销。 不同的接口提供便利的方式不同；查看接口文档的"已准备语句"。

请注意，在装载大量数据行的时候， `COPY` 几乎总是比 `INSERT` 快， 即使使用了 `PREPARE` 并且把多个 `INSERT` 命令绑在一个事务中也是这样。

当在相同事务中作为较早的 `CREATE TABLE` 或者 `TRUNCATE` 命令 使用的时候， `COPY` 是最快的。在这种情况下，没有WAL 需要写入，因为在错误情况下，这些文件包含新加载的数据将被删除。 然而，这种考虑只适用于当所有命令必须写WAL时，wal_level是 `最小的` 。

14.4.3. 删除索引

如果你正在装载一个新创建的表，最快的方法是创建表， 用 `COPY` 批量装载，然后创建表需要的任何索引。 在已存在数据的表上创建索引要比递增地更新所装载的每一行记录要快。

如果你对现有表增加大量的数据，可能先删除索引，装载表， 然后重新创建索引更快些。当然，在缺少索引的期间，其它数据库用户的数据库性能将有负面的影响。 并且我们在删除唯一索引之前还需要仔细考虑清楚，因为唯一约束提供的错误检查在缺少索引的时候会消失。

14.4.4. 删除外键约束

和索引一样，"批量地"检查外键约束比一行行检查更高效。因此，也许我们先删除外键约束，装载数据，然后重建约束会更高效。同样，装载数据和缺少约束而失去错误检查之间也有一个平衡。

更重要的是，当你将数据加载到已有外键约束的表中的时候， 每个新行需要等待触发事件的服务器列表中的项（因为它是触发器的触发，检查行的外键约束）。 装载数以百万计的行可以引起触发事件队列溢出可用内存，导致无法忍受的交换， 甚至命令的彻底失败。因此它可能是必要的，不只是理想的，当加载大量数据的时候， 删除并且重新申请外键约束。如果暂时删除约束是不能接受的， 唯一的其他资源可能会将负载操作分裂为更小的事务。

14.4.5. 增大 `maintenance_work_mem`

在装载大量的数据的时候，临时增大maintenance_work_mem配置变量可以改进性能。 这个参数也可以帮助加速 `CREATE INDEX` 和 `ALTER TABLE ADD FOREIGN KEY` 命令。 它不会对 `COPY` 本身有多大作用，所以这个建议只有在你使用上面的两个技巧时才有效。

14.4.6. 增大 `checkpoint_segments`

临时增大checkpoint_segments配置变量也可以让大量数据装载得更快。 这是因为向PostgreSQL里面装载大量的数据可以导致检查点操作 (由配置变量 `checkpoint_timeout` 声明)比平常更加频繁发生。在发生一个检查点的时候，所有脏数据都必须刷新到磁盘上。 通过在大量数据装载的时候临时增加 `checkpoint_segments` ， 所要求的检查点的数目可以减少。

14.4.7. 禁用WAL归档和流复制

当加载大量数据到使用WAL归档或流复制的安装过程时， 加载完成之后采取新的基础备份比处理大量增量WAL数据可能会更快。当加载时为防止增量WAL日志，关闭归档和流复制，通过设置 wal_level到 `最小` ， archive_mode到 `off` , max_wal_senders为零。 但是请注意，更改这些设置需要重新启动服务器。

除了为归档或者WAL发送处理WAL数据来避免时间， 这样做实际上将使某些命令更快，因为如果 `wal_level` 是 `最小的` ，他们设计不写WAL（他们可以保证碰撞安全性最后通过 `fsync` 比写WAL更便宜）。 这适用于以下命令：

- `CREATE TABLE AS SELECT`
- `CREATE INDEX` (正如 `ALTER TABLE ADD PRIMARY KEY`)
- `ALTER TABLE SET TABLESPACE`
- `CLUSTER`
- `COPY FROM` ，当目标表在同一事务之前已经被创建或截断。

14.4.8. 事后运行 `ANALYZE`

不管什么时候，如果你在更新了表中的大量数据之后，运行ANALYZE都是个好习惯。 这包含大量加载数据到表。运行 `ANALYZE` (或者 `VACUUM ANALYZE`) 可以保证规划器有表数据的最新统计。 如果没有统计数据或者统计数据太陈旧，那么规划器可能选择很差劲的查询规划， 导致表的错误或者不存在数据的性能恶化。 请注意如果启动autovacuum守护进程，可能自动运行 `ANALYZE` ；获取详情请 参阅Section 23.1.3和Section 23.1.6。

14.4.9. pg_dump的一些注意事项

pg*dump*生成的转储脚本自动使用上面的若干个技巧，但不是全部。 要尽可能快地装载*pg_dump*转储，我们需要手工做几个事情。请注意， 这些要点适用于恢复一个转储，而不是创建_一个转储的时候。 同样的要点也适用于使用psql或者pg_restore从pg_dump 归档文件装载文本复制的时候。

缺省的时候，pg_dump使用 `COPY` ，在它生成一个完整的模式和数据的转储的时候， 它会很小心地先装载数据，然后创建索引和外键。因此，在这个情况下， 头几条技巧是自动处理的。 剩余的是你要做的：

- 设置比正常状况大的 `maintenance_work_mem` 和 `checkpoint_segments` 值。
- 如果使用WAL归档或流复制，可以考虑在恢复过程中禁用他们。要做到这一点，设置 `archive_mode` 为 `off` ， `wal_level` 为 `最小的` ，并且设置 `max_wal_senders` 在装载前为零。随后，设置它们返回正确的值，并采取新的基础备份。
- 使用pg_dump和pg_restore并行转储和恢复两种模式的实验，并找到 并行作业使用的最佳数目。平行转储和恢复通过 `-j` 选项应提供给你串行模式中的更高的性能。
- 考虑整个转储是否应作为单个事务进行恢复。要做到这一点，通过 `-1` 或者 `-single-transaction` 命令行选项的psql或者pg_restore。 当使用此模式时，即使是最小的误差将回滚整个恢复，可能丢弃很多时间的处理。 取决于如何相互关联数据，这可能最好是手动清理或者没有。如果你使用单一事务并且WAL归档关闭，则 `COPY` 命令将运行速度最快。

- 如果在数据库服务器中有多个CPU，可以考虑使用pg_restore的 `-jobs` 选项。 这允许并发数据加载和索引的创建。
- 之后运行 `ANALYZE` 。

只保存数据的转储仍然会使用 `COPY` ，但是它不会删除或者重建索引， 并且它不会自动修改外键。 [1] 因此当装载只有数据的转储时候，如果你想使用这些技术， 删除以及重建索引和外键完全取决于你。 当加载数据时，增大 `checkpoint_segments` 仍然是有用的， 但是增大 `maintenance_work_mem` 就没什么必要了； 相反，你只是应该在事后手工创建索引和外键， 最后结束时不要忘记 `ANALYZE` 命令。参阅Section 23.1.3 和Section 23.1.6获取更多详情。

Notes

[1] 你可以通过使用 `-disable-triggers` 选项的方法获取关闭外键的效果。 不过要意识到这么做是消除，而不只是推迟违反外键约束， 因此如果你使用这个选项，将有可能插入坏数据。

14.5. 非持久性设置

持久性是数据库的功能，即使服务器崩溃或电源断电保证已提交事务的记录。 然而，持久性显著增加了数据库的开销， 所以如果你的网站并不需要这样的保证， PostgreSQL可以被配置保证运行更加快速。 以下是配置变化，可以在这种情况下提高性能。 除下文所述外，持久性仍然可以在数据库软件的崩溃的情况下得到保证， 只是突然操作系统停止造成数据丢失的风险或当使用这些设置时的崩溃。

- 将数据库集群的数据目录放在内存支持的文件系统中（即RAM磁盘）。 这消除了所有数据库磁盘I/O，但限制可用内存（也许交换）的数据存储量。
- 关闭fsync；没有必要删除写入磁盘的数据。
- 关闭full_page_writes；避免部分页面写入没有必要。 增加checkpoint_segments和checkpoint_timeout；这减少了检查点频率，但是增加了`/pg_xlog`的存储需求。
- 关闭synchronous_commit；这可能没有必要在每次提交时将WAL 写入磁盘。这个设置确实在只有*database*崩溃的情况下增加了事务丢失的风险（尽管没有
- 数据崩溃）。

III. 服务器管理

这部分覆盖的内容是那些 PostgreSQL 数据库管理员感兴趣的东西。 包括安装软件、设置和配置服务器、管理用户和数据库、日常维护任务。 任何运行 PostgreSQL 服务器的人，尤其是生产环境中的使用者， 都应该熟悉这部分中讨论的内容。

本部分的信息大致上是按照一个新用户的阅读顺序进行安排的。但是每个章节都是自包含的， 可以独立阅读。本部分信息是以主题单元按照陈述风格排列的。 如果读者需要查看特定命令的完整描述，那么应该看看 Part VI 。

头几章编写的风格是让那些没有前提知识的朋友也能看懂，这样那些需要架设自己的服务器的新读者就可以直接开始浏览这部分了。 其它部分是有关调节和管理的，这部分的材料是假设读者是那些经常使用 PostgreSQL 数据库系统 的读者应该熟悉的东西。我们鼓励读者阅读 Part I 和 Part II 获取额外的信息。

Table of Contents

Chapter 15. 源码安装

Table of Contents

本章讲述了如何从源代码安装PostgreSQL。 如果你安装的是预打包的版本，比如RPM或者Debian包， 那么略过这一章并阅读打包的开发人员的指导。

15.1. 简版

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

全版是本章剩余的部分。

15.2. 要求

一般说来，一个现代的与 Unix 兼容的平台应该就能运行PostgreSQL。 Section 15.6列出了到发布为止已经明确测试过的平台。 在发布的 `doc` 子目录里面有许多平台相关的FAQ， 如果你碰到问题你可能会需要参考它们。

编译PostgreSQL需要下面几样东西：

- 需要GNU make的3.80版本或者更新版本； 不能使用其它make 程序或者老版本的GNUmake。 GNU make常被安装为 `gmake` 的名字。 本文档将一直使用这个名字称呼它(在一些系统上GNUmake是名字叫 `make` 的缺省工具)。 要想测试GNUmake，敲入：

  ```
  &lt;kbd class="literal"&gt;gmake --version&lt;/kbd&gt;
  ```

- 一个ISO/ANSI C编译器（至少C89-标准）。 我们推荐使用最近版本的GCC。不过， 众所周知的是PostgreSQL可以利用许多不同厂商的不同编译器进行编译。

- 除了gzip或者bzip2之外，还需要tar来解开发布。

- 缺省时将自动使用GNU Readline库 它允许psql（PostgreSQL命令行SQL解释器）记住每个键入的命令， 并允许你使用箭头键来调用和编辑以前的命令。这是非常有帮助的，强烈推荐。 如果你不想用它，那么你必需给 `configure` 声明 `--without-readline` 选项。 作为替代，你可以经常使用BSD许可 `libedit` 的库，最初 在NetBSD上开发的。 如果没有发现 `libreadline` ， 或者如果为 `configure` 指定 `--with-libedit-preferred` ， 可以使用与GNU Readline兼容的 `libedit` 库。 如果你使用的是一个基于包的 Linux 发布，那么要注意你需要 `readline` 和 `readline-devel` 两个包， 特别是如果这两个包在你的版本里是分开的时候。

- 缺省的时候将使用zlib压缩库。如果你不想使用它， 那么你必须给 `configure` 声明 `--without-zlib` 选项。 使用这个选项关闭了在pg_dump和 pg_restore里面压缩归档的支持。

下列包是可选的。在缺省配置的时候并不要求它们， 但是如果打开了一些编译选项之后就需要它们了，如下面解释的:

- 要编译PL/Perl服务器端编程语言，你需要一个完整的Perl安装， 包括 `libperl` 库和头文件。因为PL/Perl是一个共享库， `libperl` 库在大多数平台上也必须是一个共享库。 最新版本的Perl好像已经是缺省这么做了，但是早先的版本可不是这样的， 而且在任何安装了Perl的地方总是可选的。 如果你打算比偶尔使用PL/Perl编译更多， 你应确保编译带有 `usemultiplicity` 启用选项的Perl安装 （ `perl -V` 将显示是否是这种情况）。

如果没有共享库，但却需要它，那么在PostgreSQL编译过程中将看到下面的信息，指出这个问题：

```
*** Cannot build PL/Perl because libperl is not a shared library.
*** You might have to rebuild your Perl installation.  Refer to
*** the documentation for details.
```

如果你不按照屏幕输出的指示去做，那么你只会注意到PL/Perl库对象 `plperl.so` 或者类似的什么东西， 不会安装到系统里。如果你看到这些东西，那么你就必须重新手工编译并安装Perl， 这样才能编译PL/Perl。在配置Perl的过程中，要求一个共享库。

- 要编译PL/Python服务器端编程语言，你需要一个Python的安装， 包括头文件和distutils模块。 最小所需版本是Python2.3。 如果版本是3.1或者更高版本，则支持Python 3；但是当使用Python 3的时候， 则参阅 Section 43.1。

  因为PL/Python将以共享库的方式编译， `libpython` 库在大多数平台上也必须是一个共享库。 在缺省的Python安装时不是这样的。如果在编译和安装PostgreSQL之后， 你有一个叫做 `plpython.so` 的文件(可能扩展名会有所不同)，那么一切都好说， 否则你应该会看到类似下面的信息飘过:

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation.  Refer to
  *** the documentation for details.
  ```

  这意味着你必须重新编译(一部分)Python安装，以创建这个共享库。

  如果有问题，用 `--enable-shared` 标志运行Python 2.3或更高版本的configure 脚本。 在有些操作系统上，你不必非要编译一个共享库，不过你需要让PostgreSQL的编译系统知道这些。 参考 `src/pl/plpython` 目录中的 `Makefile` 获取细节。

- 如果想编译PL/Tcl过程语言，那么当然需要安装Tcl了。 如果你使用Tcl先前的8.4版本，应该确保它不需要多线程支持。

- 要打开本地语言支持(NLS)，也就是说，用英语之外的语言显示程序的信息， 你需要一个Gettext API的实现。 有些操作系统内置了这些(比如Linux, NetBSD, Solaris)， 对于其它系统，你可以从 http://www.gnu.org/software/gettext/下载一个额外的包。 如果你在GNU C库里面使用Gettext实现， 那么你就额外需要GNU Gettext包， 因为我们需要里面的几个工具程序。对于任何其它的实现，你应该不需要它。

- 你需要Kerberos, OpenSSL, OpenLDAP,和/或者PAM， 如果你想支持使用这些服务的认证或者加密，那你需要这些包。

- 为了编译PostgreSQL文档，有一套独立要求；参阅 Section J.2.

如果你从Git树中编译，而不是使用发布的源代码包， 或者你想做一些服务开发， 那么你还需要下面的包：

- 如果你需要从Git校验中编译，或者你修改了扫描器和分析器的定义文件， 那么你需要Flex和Bison。如果你需要它们， 那么确保自己拿到的是Flex 2.5.31或更新的版本，以及Bison 1.875或者更新的版本。 其它的lex和yacc程序肯定是不行的。
- 如果需要从Git校验中编译或者如果需要改变使用Perl脚本的任何编译步骤的输入文件，那么 需要Perl 5.8或者更新版本。如果在Windows上编译，那么你在任何情况下将需要Perl。

如果你需要获取GNU包，你可以在GNU镜像站点 http://www.gnu.org/order/ftp.html或者ftp://ftp.gnu.org/gnu/ 找到它们。

请检查一下，看看你是否有足够的磁盘空间。 你将大概需要近100MB 用于存放安装过程中的源码树和大约20MB用于安装目录。 一个空数据库大概需要35MB。 然后在使用过程中大概需要在一个平面文本文件里存放同等数据量五倍的空间存储数据。 如果你要运行回归测试，还临时需要额外的150MB 。 请用 `df` 命令检查剩余磁盘空间。

15.3. 获取源码

PostgreSQL 9.3.1，源码可以从我们的网站: http://www.postgresql.org/download/ 进行下载。你可以获得文件命名 `postgresql-9.3.1.tar.gz` 或者 `postgresql-9.3.1.tar.bz2` 。在你获取文件之后，解压缩：

```
<kbd class="literal">gunzip postgresql-9.3.1.tar.gz</kbd>
<kbd class="literal">tar xf postgresql-9.3.1.tar</kbd>
```

如果你有 `.bz2` 文件，可以使用 `bunzip2` 代替 `gunzip` 。 这样将在当前目录创建一个目录 `postgresql-9.3.1` ， 里面是PostgreSQL源代码。进入这个目录完成安装过程的其它步骤。

你也可以直接从版本控制库中获取源码，参阅Appendix I。

15.4. 安装过程

1. 配置

   安装过程的第一步就是配置源代码树并选择你喜欢的选项。这个工作是通过运行 `configure` 脚本实现的， 对于缺省安装，你只需要简单地敲入:

   ```
   &lt;kbd class="literal"&gt;./configure&lt;/kbd&gt;
   ```

   该脚本将运行一些测试来决定一些系统相关的变量，并检测操作系统的设置， 最后将在编译树中创建一些文件以记录它找到了什么。 如果你想保持编译目录的独立， 那么你也可以在源代码树之外的其它目录里运行 `configure` 。 这个过程也被称为*VPATH* 编译。这里是方法：

   ```
   &lt;kbd class="literal"&gt;mkdir build_dir&lt;/kbd&gt;
   &lt;kbd class="literal"&gt;cd build_dir&lt;/kbd&gt;
   &lt;kbd class="literal"&gt;/path/to/source/tree/configure [options go here]&lt;/kbd&g
   &lt;kbd class="literal"&gt;gmake&lt;/kbd&gt;
   ```

   缺省设置将编译服务器和应用程序，还有所有只需要C编译器的客户端程序和接口。 缺省时所有文件都将安装到 `/usr/local/pgsql` 目录。

   你可以通过给出下面的一个或多个 `configure` 命令 行选项来自定义编译和安装过程：

   --prefix=``_PREFIX_

   把所有文件装在 `_PREFIX_` 目录下而不是 `/usr/local/pgsql` 里。 实际的文件会安装到不同的子目录里； 甚至没有一个文件会直接安装到 `_PREFIX_` 目录里。

   如果你有特殊需要，你还可以用下面的选项自定义不同子目录的位置。 不过，即使你保持缺省设置，安装时浮动的， 意味着可以在安装之后移动目录(`man` 和 `doc` 位置不受影响)。

   为了浮动安装，你可能需要使用 `configure` 的 `--disable-rpath` 选项。 还有，你需要告诉操作系统如何找到共享库。

   --exec-prefix=``_EXEC-PREFIX_

   把体系相关的文件安装到 `_EXEC-PREFIX_` ， 而不是 `_PREFIX_` 设置的地方。 这样做可以比较方便地在不同主机之间共享体系相关的文件。如果你省略它， 那么 `_EXEC-PREFIX_` 就会被设置为等于 `_PREFIX_` 并且体系相关和体系无关的文件 都会安装到同一目录树下， 这也可能是你想要的。

   --bindir=``_DIRECTORY_

声明可执行程序的目录，缺省是 `_EXEC-PREFIX_` /bin， 通常也就是 `/usr/local/pgsql/bin` 。

`--sysconfdir=``_DIRECTORY_`

设置各种配置文件的目录。缺省是 `_PREFIX_` /etc。

`--libdir=``_DIRECTORY_`

设置库文件和动态装载模块的目录。缺省是 `_EXEC-PREFIX_` /lib。

`--includedir=``_DIRECTORY_`

设置C和C++头文件的目录。缺省是 `_PREFIX_` /include。

`--datarootdir=``_DIRECTORY_`

设置各种类型只读数据文件的根目录。这只设置了一些下面的缺省选项。缺省是 `_PREFIX_` /share。

`--datadir=``_DIRECTORY_`

设置使用安装程序的只读数据文件的目录，缺省是 `_DATAROOTDIR_` 。 请注意这与你的数据库文件放在哪里无关。

`--localedir=``_DIRECTORY_`

设置安装现场数据的目录，特别是消息转变目录文件。缺省是 `_DATAROOTDIR_` /locale。

`--mandir=``_DIRECTORY_`

随着PostgreSQL一起的手册页将安装到这个目录的 `man``_x_` 子目录里。 缺省是 `_DATAROOTDIR_` /man。

`--docdir=``_DIRECTORY_`

设置除"man"以外的文档文件的根目录。这只设置下面选项的缺省， 这个选项的缺省值是 `_DATAROOTDIR_` /doc/postgresql。

`--htmldir=``_DIRECTORY_`

PostgreSQL的HTML-格式文档将安装在这个目录。 缺省是 `_DATAROOTDIR_` 。

> **Note:** 为了让PostgreSQL能够安装在一些共享的安装位置(比如 `/usr/local/include`)，同时又不至于和系统其它部分产生名字空间干扰，我们采取了一些步骤。 首先，安装脚本会自动给 `datadir` ， `sysconfdir` 和 `docdir` 后面附加上" `/postgresql` "字符串， 除非展开的完整路径名已经包含字符串" `postgres` "或者 " `pgsql` "。 比如，如果你选择 `/usr/local` 作为前缀， 那么文档将安装在 `/usr/local/doc/postgresql` ， 但如果前缀是 `/opt/postgres` ， 那么它将被放到 `/opt/postgres/doc` 。 客户接口的公共C头文件安装到了 `includedir` ， 并且是名字空间无关的。 内部的头文件和服务器头文件都安装

到 `includedir` 下的私有目录中去了。 参考每种接口的文档获取关于如何访问头文件的信息。最后，如果合适， 那么也会在 `libdir` 下创建一个私有的子目录，用于动态装载模块。

`--with-includes=``_DIRECTORIES_`

`_DIRECTORIES_` 是一系列冒号分隔的目录， 这些目录将被加入编译器的头文件搜索列表中。 如果你有一些可选的包(比如GNU Readline)安装在非标准位置， 你就必须使用这个选项， 以及可能还有相应的 `--with-libraries` 选项。

例子： `--with-includes=/opt/gnu/include:/usr/sup/include` 。

`--with-libraries=``_DIRECTORIES_`

`_DIRECTORIES_` 是一系列冒号分隔的目录，这些目录是用于查找库文件的。 如果你有一些包安装在非标准位置， 你可能就需要使用这个选项(以及对应的 `--with-includes` 选项)。

例子： `--with-libraries=/opt/gnu/lib:/usr/sup/lib` 。

`--enable-nls[=``_LANGUAGES_`]

打开本地语言支持(NLS)，也就是以非英文显示程序信息的能力。 `_LANGUAGES_` 是一个可选的空格分隔的语言代码列表，标识你想支持的语言。 比如 `--enable-nls='de fr'` 。你提供的列表和实际支持的列表之间的交集会自动计算出来。 如果你没有声明一个列表，那么就安装所有可用的翻译。

要使用这个选项，你需要一个Gettext的实现。 见上文。

`--with-pgport=``_NUMBER_`

`_NUMBER_` 为服务器和客户端的缺省端口(缺省是 5432)。这个端口可以在以后设置， 不过如果你在这里声明，那么服务器和客户端将有相同的编译好了的缺省值。这样会方便些。 通常选取一个非缺省值的好理由是你企图在同一台机器上运行多个PostgreSQL服务器。

`--with-perl`

编译PL/Perl服务器端编程语言。

`--with-python`

编译PL/Python服务器端编程语言。

`--with-tcl`

编译PL/Tcl服务器端编程语言。

`--with-tclconfig=``_DIRECTORY_`

Tcl安装的 `tclConfig.sh` 文件所在目录，它里面包含编译 Tcl 模块的配置信息。 这个文件通常会自动在约定俗成的位置找到这些文件，但是如果你需要一个不同版本的Tcl， 你也可以声明能找到它的目录。

`--with-gssapi`

编译支持GSSAPI认证的东西。在许多系统上，GSSAPI（通常Kerberos安装部分）系统没有安装在缺省的搜索目录下 (比如 `/usr/include` , `/usr/lib`)， 所以你必须使用附加的 `--with-includes` 和 `--with-libraries` 选项。 `configure` 在继续配置之前将检查所需要的头文件和库， 以确保GSSAPI是充分可用的。

`--with-krb5`

编译支持 Kerberos 5 认证的东西。在许多系统上， Kerberos 系统没有安装在缺省的搜索目录下(比如 `/usr/include` , `/usr/lib`)， 所以你必须使用附加的 `--with-includes` 和 `--with-libraries` 选项。 `configure` 在继续配置之前将检查所需要的头文件和库，以确保 Kerberos 是充分可用的。

`--with-krb-srvnam=``_NAME_`

缺省的Kerberos服务主的名称（通过GSSAPI使用）。缺省是 `postgres` 。 通常没有理由改变这个值。除非你在Windows环境下，在这种情况下必须设置大写 `POSTGRES` 。

`--with-openssl`

编译支持SSL(加密的)连接。这个选项需要安装OpenSSL包。 `configure` 将在安装之前检查所需要的头文件和库文件以确信OpenSSL安装是充分可用的。

`--with-pam`

编译PAM(可插拔认证模块)支持。

`--with-ldap`

编译LDAP支持。 用于认证和查找连接参数(参见Section 31.17和Section 19.3.8以获取更多信息)。 在 Unix 上，这需要OpenLDAP包的支持。 在Windows上，缺省使用WinLDAP库。 `configure` 将会检查所需的头文件和库以确保OpenLDAP的安装是充分可用的。

`--without-readline`

避免使用Readline与libedit库。 这样就关闭了psql里的命令行编辑和历史，因此我们不建议这么做。

`--with-libedit-preferred`

优先使用BSD-认证的libedit库而不是GPL认证的Readline库。 该选项仅在同时安装了这两个库的情况下才有意义。 缺省使用Readline库。

`--with-bonjour`

编译Bonjour支持。这要求操作系统支持Bonjour。在Mac OS X上建议使用。

`--with-ossp-uuid`

使用OSSP UUID library编译组件。 特别是，编译 uuid-ossp模块， 它提供了函数产生UUIDs.。

`--with-libxml`

编译libxml (开启SQL/XML支持)。需要Libxml 2.6.23或者更高版本支持这一特性。

Libxml安装程序 `xml2-config` 可用于检测所需的编译器和链接器选项。 如果发现，PostgreSQL将自动使用它。 在一个不寻常的位置来指定libxml安装， 您可以要么设置环境变量 `XML2_CONFIG` 以指向 `xml2-config` 附属于安装的程序 或者使用选项 `--with-includes` 和 `--with-libraries` 。

`--with-libxslt`

当编译 xml2模块时，使用libxslt。 xml2依赖于这个库执行XSL转变成XML。

`--disable-integer-datetimes`

禁用64 位的时间戳整数存储和时间间隔支持。并且作为浮点数存储时间值。 在PostgreSQL发布之前的8.4版本中缺省浮点日期时间存储。 但它现在已经过时，因为它没有支持 `timestamp` 值的全范围的微秒精度。 然而，基于整数的日期时间存储需要一个64位的整数类型。 因此，当没有这样的类型可用时，可以使用此选项时， 或与PostgreSQL先前版本编写的应用程序兼容。 参阅 Section 8.5获取更多的信息。

`--disable-float4-byval`

禁用"按值传递"float4值，使它们"通过引用"传递。 此选项消耗性能，但可能需要与用C语言编写的旧的用户定义的函数兼容，并且使用 "版本 0"调用约定。一个更好的长期的解决办法是使用"版本 1"调用约定更新任何此类函数。

`--disable-float8-byval`

禁用"按值传递"传递float8值，使它们"通过引用"传递。 此选项消耗性能，但可能需要与用C语言编写的旧的用户定义的函数兼容，并且使用 "版本 0"调用约定。一个更好的长期的解决办法是使用"版本 1"调用约定更新任何此类函数 请注意，此选项不仅影响float8，而且也影响int8以及一些相关类型（如时间戳）。 在32位平台上，缺省是 `--disable-float8-byval` 。 并且它不允许选择 `--enable-float8-byval` 。

`--with-segsize=``_SEGSIZE_`

设置段大小， 以GB为单位。 大表被分成多个操作系统文件，每个文件的大小等同于段大小。 这避免了存在许多平台上的文件大小限制。 默认段大小，1千兆字节，在所有支持的平台上是安全的。 如果你的操作系统支持"largefile"（现在大多数支持）， 你可以使用较大的段大小。 这可以帮助减少当大表工作时消耗掉的文件描述符数量。 但要小心，不要选择一个大于通过您的平台和您打算使用的文件系统支持的值， 你可能希望使用其他工具，比如tar， 也可以设置可用文件大小的限制。 因此建议，虽然不是绝对必要的，但该值是2的幂。 请注意，改变这个值需要初始化数据库。

`--with-blocksize=``_BLOCKSIZE_`

设置块大小，以KB为单位。这是表中存储和I/O单元。 默认情况下，8千字节，适用于大多数情况； 但是其它的值可能在特殊情况下是有用的。 该值必须是1和32之间（千字节）2的幂。 请注意，改变这个值需要初始化数据库。

`--with-wal-segsize=``_SEGSIZE_`

设置*WAL*段大小，以MB为单位。 这是WAL日志中每个单独的文件的大小。它可能有助于调整这个大小来控制WAL日志传送的粒度。 缺省大小为16兆字节。 该值必须是1和64之间（兆字节）2的幂。 请注意，改变这个值需要初始化数据库。

`--with-wal-blocksize=``_BLOCKSIZE_`

设置*WAL*块大小，以KB为单位。 这是WAL日志中存储和I/O单元。 默认情况下，8千字节，适用于大多数情况； 但是其它的值可能在特殊情况下是有用的。 该值必须是1和64之间（千字节）2的幂。 请注意，改变这个值需要初始化数据库。

`--disable-spinlocks`

允许在PostgreSQL没有该平台的CPU自旋锁支持的情况下编译成功。 缺乏自旋锁的支持将导致性能恶化； 因此，只有在编译过程退出，并且告诉你说该平台缺乏自旋锁支持的时候才使用这个选项。 如果在你的平台上需要这个选项才能编译PostgreSQL， 请向PostgreSQL开发者报告这个问题。

`--disable-thread-safety`

禁用客户端库是线程安全的。 这样就允许在libpq和 ECPG程序里的并发线程安全地控制他们私有的连接句柄。

`--with-system-tzdata=``_DIRECTORY_`

PostgreSQL包括它自己的时区数据库， 它要求对日期和时间操作。此时区数据库实际上是与许多操作系统比如FreeBSD,Linux, and Solaris提供的"zoneinfo"时区数据库兼容。所以重新安装将是多余的。当这个选项被使用时， `_DIRECTORY_` 中系统提供的时区数据库是用来代替包含在PostgreSQL源代码发布中的其中之一。 `_DIRECTORY_` 必须作为绝对路径被指定。 `/usr/share/zoneinfo` 是某些操作系统上的可能目录。 请注意，这个安装程序将不检测不匹配的或者错误的时间区域数据。 如果您使用此选项，建议您运行回归测试，以验证该时区的数据， 你已经指出正确使用PostgreSQL。

该选项主要是针对二进制软件包分发者，他知道目标操作系统运行良好。 使用此选项的主要优势是，当许多当地白昼节约时间规则变化的任何时候，PostgreSQL包不需要升级。 另一个优点是，如果时区数据库文件在安装期间不需要被编译，PostgreSQL可以进行交叉编译更直截了当。

`--without-zlib`

避免使用Zlib库。这样就关闭了pg_dump 和pg_restore里面的压缩支持。 这个选项只适用于那些没有这个库的罕见的系统。

`--enable-debug`

把所有程序和库以带有调试符号的方式编译。这意味着你可以通过一个调试器运行程序来分析问题。 这样做显著增大了最后安装的可执行文件的大小，并且在非GCC的编译器上， 这么做通常还要关闭编译器优化，导致速度的下降。但是，如果有这些符号表的话， 就可以非常有效地帮助定位可能发生问题的位置。目前， 我们只是在你使用GCC的情况下才建议在生产安装中使用这个选项。 但是如果你正在进行开发工作，或者正在使用beta版本， 那么你就总应该打开它。

`--enable-coverage`

如果使用GCC，所有的程序和库连同代码覆盖测试仪器一起被编译。 在运行时，它们与代码覆盖率度量在编译目录下生成文件。 参阅Section 30.4 获取更多信息当做开发工作时，该选项仅用于GCC。

`--enable-profiling`

如果使用GCC，则编译所有程序和库，使他们可以描绘轮廓。在后端出口，创建子目录， 包含分析使用的 `gmon.out` 文件。 当做开发工作时，该选项仅用于GCC。

`--enable-cassert`

打开服务器中的*assertion*检查，它会检查许多"不可能发生"的条件。 它对于代码开发的用途而言是无价之宝，不过这些测试可以显著减缓服务器。 并且，打开这个测试不会提高系统的稳定性！这些断言检查并不是按照错误的严重性分类的， 因此一些相对无害的小虫子也可能导致服务器重启 (只要它触发了一次断言失败)。目前， 我们不推荐在生产环境中使用这个选项，但是如果你在做开发或者在使用beta版本的时候应该打开它。

`--enable-depend`

打开自动倚赖性跟踪。如果打开这个选项， 那么 makefile 文件将设置为在任何头文件被修改的时候都将重新编译所有受影响的目标文件。 如果你在做开发的工作，那么这个选项很有用，但是如果你只是想编译一次并且安装， 那么这就是浪费时间。目前，这个选项只有在你使用GCC的时候才管用。

`--enable-dtrace`

编译PostgreSQL支持动态跟踪工具DTrace。 参阅Section 27.4获取更多信息。

指向 `dtrace` 程序，设置环境变量 `DTRACE` 。 这往往是必须的，因为 `dtrace` 通常安装在 `/usr/sbin` 中且该目录一般不在搜索路经中。

在环境变量 `DTRACEFLAGS` 中为 `dtrace` 程序指定 额外命令行选项。在Solaris上， 要包含64位二进制的DTrace支持，需要指定 `DTRACEFLAGS="-64"` ，比如， 在使用GCC编译的时候:

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

在使用Sun编译器的时候：

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS='
```

如果你喜欢使用不同于 `configure` 找出来的 C 编译器， 可以将环境变量 `CC` 设置为你选择的程序。缺省时， `configure` 将选择 `gcc` (只要可用)， 或者是该平台的缺省(通常是 `cc`)类似地， 你可以用 `CFLAGS` 覆盖缺省编译器标志。

你可以在 `configure` 命令行上声明环境变量，比如：

```
&lt;kbd class="literal"&gt;./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'&lt;/kbd&gt;
```

下面是可用的环境变量列表：

`BISON`

Bison程序

`CC`

C编译器

`CFLAGS`

传递给C编译器的选项

`CPP`

C预处理器

`CPPFLAGS`

传递给C预处理器的选项

`DTRACE`

`dtrace` 程序位置

`DTRACEFLAGS`

传递给 `dtrace` 程序的选项

`FLEX`

Flex程序

`LDFLAGS`

当连接可执行或可共享库时使用的选项

`LDFLAGS_EX`

只连接可执行时的额外选项

`LDFLAGS_SL`

只连接共享库时的额外选项

`MSGFMT`

本地语言支持的 `msgfmt` 程序

`PERL`

Perl解释器的完整路径。用于确定编译PL/Perl的依赖关系

`PYTHON`

Python解释器的完整路径。用于确定编译PL/Python的依赖关系。 另外，Python 2或3是否在这声明（或另有隐式选择）决定了PL/Python 的哪种语言是可用的。 参阅Section 43.1 获取更多详情。

`TCLSH`

Tcl解释器的完整路径。用于确定编译PL/Tcl的依赖关系。并且它将替代Tcl脚本。

`XML2_CONFIG`

`xml2-config` 程序用于定位libxml安装。

> **Note:** 当开发服务器内部代码时，建议使用配置选项 `--enable-cassert` （其中 打开许多运行时错误检查）和 `--enable-debug` （可以改进调试工具的有效性）。 > > 如果使用GCC，最好是进行至少 `-O1` 的优化级别的编译。 因为不使用优化(`-O0`)禁止一些重要的编译器警告（例如，使用未初始化变量）。 然而，非零优化级别可以进行复杂调试， 因为通过编译代码步进调试往往不会匹配一对之一源代码行。 如果你感到困惑，而试图调试优化的代码，重新编译 `-O0` 的指定文件。 一个简单方法来做到这一点是通过传递选项到make: `gmake PROFILE=-O0 file.o` 。

2. 编译

开始编译，键入：

```
&lt;kbd class="literal"&gt;gmake&lt;/kbd&gt;
```

（记住使用GNU make）。 依硬件不同，编译过程可能需要一些时间。显示的最后一行应该是：

```
All of PostgreSQL is successfully made. Ready to install.
```

如果你想要编译一切可以编译的东西，包含文档（HTML和帮助手册），以及额外模块（ `contrib` ） 键入：

```
&lt;kbd class="literal"&gt;gmake world&lt;/kbd&gt;
```

显示的最后一行应该是：

```
PostgreSQL, contrib and HTML documentation successfully made. Ready to install.
```

3. 回归测试

   如果你想在安装文件前测试新编译的服务器， 那么你可以在这个时候运行回归测试。 回归测试是一个用于验证PostgreSQL在系统上是否按照开发人员设想的那样运行的测试套件。 敲入：

   ```
   &lt;kbd class="literal"&gt;gmake check&lt;/kbd&gt;
   ```

   这条命令在root里无法使用；请在非特权用户下运行该命令。 Chapter 30 包含 关于如何解释测试结果的详细信息。 你可以在以后的任何时间通过执行这条命令来运行这个测试。

4. 安装文件

   > **Note:** 如果你正在升级一套现有的系统必定读 Section 17.6。 它有关于升级集群的说明。

   安装PostgreSQL，键入：

   ```
   &lt;kbd class="literal"&gt;gmake install&lt;/kbd&gt;
   ```

   这条命令将把文件安装到在step 1声明的目录里面去。 确保你对那个目录有足够的权限。通常你需要用 root 权限做这一步。 或者你也可以事先创建目标目录并且分派合适的权限。

   安装文档（HTML和手册页），键入：

   ```
   &lt;kbd class="literal"&gt;gmake install-docs&lt;/kbd&gt;
   ```

   如果上面编译了world,而不是键入：

   ```
   &lt;kbd class="literal"&gt;gmake install-world&lt;/kbd&gt;
   ```

   这也安装文档。

你可以使用 `gmake install-strip` 代替 `gmake install` 在安装可执行文件和库文件时把它们的调试信息抽取掉。 这样将节约一些空间。如果你编译时带着调试支持，那么抽取将有效地删除调试支持， 因此我们应该只是在不再需要调试的时候做这些事情。 `install-strip` 力图做一些合理的事情来节约空间， 但是它并不知道如何从可执行文件中抽取每个不需要的字节， 因此，如果你希望节约所有可能节约的磁盘空间，那么你可能需要手工做些处理。

标准的安装只提供所有开发客户端应用的头文件和服务器端的程序开发， 比如用 C 写客户函数或者数据类型的头文件。 （先于PostgreSQL 8.0，后者需要单独的 `gmake install-all-headers` 命令，但是标准安装中已经包含这一步）。

只装客户端: 如果你只想装客户应用和接口，那么你可以用下面的命令:

```
&lt;kbd class="literal"&gt;gmake -C src/bin install&lt;/kbd&gt;
&lt;kbd class="literal"&gt;gmake -C src/include install&lt;/kbd&gt;
&lt;kbd class="literal"&gt;gmake -C src/interfaces install&lt;/kbd&gt;
&lt;kbd class="literal"&gt;gmake -C doc install&lt;/kbd&gt;
```

`src/bin` 中有一些仅供服务器使用的二进制文件，但是它们都很小。

卸载: 可以使用 `gmake uninstall` 命令卸载。 不过这样不会删除任何创建出来的目录。

清理: 在安装完成以后，你可以通过在源码树里面用命令 `gmake clean` 删除编译过程文件释放磁盘空间。 这样会保留 `configure` 程序生成的文件，这样以后你就可以用 `gmake` 命令重新编译所有东西。 要把源码树恢复为发布时的状态，用 `gmake distclean` 命令。 如果你想从同一棵源码树上为多个不同平台编译， 你就一定要运行这条命令并且为每个平台重新配置。 另外，在每种系统上使用一套独立的编译树，这样源代码树就可以保留不被更改。

如果你执行了一次编译，然后发现你的 `配置` 选项是错误的， 或者你修改了任何 `配置` 所探测的东西(比如升级了软件)， 那么在重新配置和编译之前运行一下 `gmake distclean` 是个好习惯。如果不做这个事情， 你修改的配置选项可能无法传播到所有需要变化的地方。

15.5. 安装后设置

15.5.1. 共享库

在一些有共享库的系统里， 你需要告诉系统如何找到新安装的共享库。 那些并不是必须做这个工作的系统包括 FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (以前的 Digital UNIX)和 Solaris。

设置共享库的搜索路径的方法因平台而异， 但是最广泛使用的方法是设置 `LD_LIBRARY_PATH` 环境变量， 比如在Bourne shell(`sh` , `ksh` , `bash` , `zsh`)中：

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

在 `csh` 或者 `tcsh` 中：

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

把 `/usr/local/pgsql/lib` 换成你在step 1设置的 `--libdir` 。 你应该把这些命令放到启动文件，如 `/etc/profile` 或者 `~/.bash_profile` 里面。 和这个方法相关的一些注意事项和很好的信息可以在http://xahlee.org/UnixResource*dir*/ldpath.html找到。

在有些系统上，更好的方法可能是在编译之前设置 `LD_RUN_PATH` 环境变量。

在Cygwin里， 把库目录放在 `PATH` 或者把 `.dll` 文件移动到 `bin` 目录。

如果有疑问， 请参考系统的手册页(可能是 ld.so 或 rld)。如果稍后你收到下面这样的信息：

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

那么这一步就是必须的了。只需关注一下就是了。

如果你的系统是Linux并且你还有root权限， 那么你可以运行：

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(或者相应的目录)以便让运行时链接器更快地找到共享库。 请参考 `ldconfig` 的手册页获取更多信息。 在FreeBSD, NetBSD和OpenBSD上，命令是：

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

我们不知道其它系统有什么等效命令。

15.5.2. 环境变量

如果你安装到 `/usr/local/pgsql` 或者其它什么缺省时不搜索程序的地方， 那你应该增加一个 `/usr/local/pgsql/bin` (或者是你在step 1中给你的 `PATH` 设置 `--bindir` 的值)。 严格说，这些都不是必须的， 但这么做可以让你使用PostgreSQL更方便。

要做这些事情，把下面几行加到shell启动文件， 如 `~/.bash_profile` 或者 `/etc/profile` (如果你想影响所有用户)：

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

如果你用的是 `csh` 或者 `tcsh` ，那么用这条命令：

```
set path = ( /usr/local/pgsql/bin $path )
```

为了让系统找得到man文档， 你需要加类似下面的一行到一个shell启动文件里(除非你安装到了缺省时搜索的位置)：

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

环境变量 `PGHOST` 和 `PGPORT` 为客户端应用声明了数据库服务器的主机和端口， 覆盖了那些编译时的缺省项。如果你想从远端运行客户端应用， 那么每个准备使用该数据库的用户都设置 `PGHOST` 将会非常方便。 但这不是必须的，而且大部分客户端程序也可以通过命令行选项替换这些设置。

15.6. 支持平台

平台（即CPU架构和操作系统组合）被认为是由PostgreSQL开发社区支持的。 如果代码包含在工作平台上的归定，并且它最近被证实编译，在该平台上传递其回归测试。 目前，平台兼容性的大多数测试通过PostgreSQL Build Farm上的试验机自动完成。 如果您有兴趣在一个未出现bulid farm平台上使用PostgreSQL， 但对其中代码工作位置或者可以进行工作，我们强烈建议你建立一个bulid farm成员机器， 这样可以保证持续兼容性。

在一般情况下，PostgreSQL可以预期工作在这些CPU架构上：x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K, 和PA-RISC。存在M32R, NS32K和VAX代码的支持， 但这些体系结构并不知道最近已经测试过。它往往通过配置 `--disable-spinlocks` 编译不支持的CPU类型。 但性能会很差。

PostgreSQL预期工作在这些操作操作系统上： Linux（所有近期发行的）， Windows（Win2000 SP4和更高版本）， FreeBSD, OpenBSD, NetBSD, Mac OS X, AIX, HP/UX, IRIX, Solaris, Tru64 Unix, 和UnixWare。其它Unix系统也可以工作，但目前尚未被测试。 在大多数情况下，所有支持给定操作系统的CPU架构将工作。 查看Section 15.7，看是否 有声明您的操作系统的信息，特别是如果使用的是旧系统。

如果你有根据最近编译结果支持的平台安装问题，请报告给 `<[pgsql-bugs@postgresql.org](mailto:pgsql-bugs@postgresql.org)>` 。 如果你有兴趣移植PostgreSQL到一个新的平台， `<[pgsql-hackers@postgresql.org](mailto:pgsql-hackers@postgresql.org)>` 是讨论这类问题的合适地方。

15.7. 特定平台注意事项

这部分提供额外的关于PostgreSQL的安装和设置的特定平台的问题。 一定要阅读安装说明，尤其是Section 15.2。 另外，检查 Chapter 30有关回归测试结果的说明。

此处未覆盖的平台没有已知的特定平台安装问题。

15.7.1. AIX

PostgreSQL工作在AIX上，但正确安装它具有挑战性。 支持AIX 4.3.3到6.1版本。 您可以使用GCC或本机IBM编译器 `xlc` 。 一般情况下，采用近期的AIX版本和PostgreSQL有所帮助。 检查bulid farm获得最新的信息，其中AIX版本是众所周知的。

推荐的最小修复水平支持的AIX版本是：

AIX 4.3.3

Maintenance Level 11 + post ML11 bundle

AIX 5.1

Maintenance Level 9 + post ML9 bundle

AIX 5.2

Technology Level 10 Service Pack 3

AIX 5.3

Technology Level 7

AIX 6.1

Base Level

查看当前补丁级别，使用AIX 4.3.3到AIX 5.2 ML7中的 `oslevel -r` ， 或者更高版本的 `oslevel -s` 。

除你自己之外请使用以下的 `configure` 标志， 如果你在 `/usr/local` ： `--with-includes=/usr/local/include --with-libraries=/usr/local/lib` 中已经安装了Readline或者libz：

15.7.1.1. GCC问题

在AIX 5.3上，PostgreSQL的编译以及使用GCC运行有一些问题。

您将要使用GCC随后到3.3.2的版本，特别是如果你使用一个预先包装的版本。 我们在4.0.1时有很大成功。比起GCC的实际问题来说，早期版本问题似乎与IBM包装GCC有更大关系，因此，如果你自己编译GCC，关于GCC早期版本你很可能成功。

15.7.1.2. Unix-域套接字中断

AIX 5.3有问题，其中 `sockaddr_storage` 没有被定义为足够大。 在5.3版本中，IBM增加 `sockaddr_un` 的大小， Unix域套接字地址结构，但并没有相应地增加 `sockaddr_storage` 的大小。 这一结果是尝试使用Unix域套接字与PostgreSQL导致libpq溢出的数据结构。 TCP/IP连接工作正常，但不是Unix域套接字，以防止工作着的回归测试。

这个问题报道给IBM，并且作为bug报告PMR29657进行记录。 如果您升级到维护级别5300-03或更高版本，这将包括此修复程序。 一个快速解决方法是改变 `/usr/include/sys/socket.h` 中的 `_SS_MAXSIZE` 到1025。在这两种情况下， 一旦你有修正的头文件，则要重新编译PostgreSQL。

15.7.1.3. 网络地址问题

PostgreSQL依赖于系统的 `getaddrinfo` 函数解析 `listen_addresses` ， `pg_hba.conf` 中的IP地址， 较旧版本的AIX已经划分出这个函数中的bug。 如果您有关于这些设置的问题， 更新到上面显示的相应的AIX补丁级别应特别注意。

一个用户报告：

当在AIX 5.3上实现PostgreSQL版本8.1的时候， 我们周期性地遇到了问题， 其中统计收集器可能"诡秘的"不会成功。 这似乎是IPv6实行中意外行为的结果。 它看起来像PostgreSQL和IPv6在AIX 5.3中不能很好地结合。

任何下面操作"修复"这个问题。

- 删除本地主机的IPv6地址：

  ```
  (as root)
  # ifconfig lo0 inet6 ::1/0 delete
  ```

- 从网络服务中删除IPv6。在AIX上的文件 `/etc/netsvc.conf` 大致 等同于Solaris/Linux上的 `/etc/nsswitch.conf` 。 默认情况下，AIX上是这样的:

  ```
  hosts=local,bind
  ```

  替换为：

  ```
  hosts=local4,bind4
  ```

解除搜索IPv6地址。

> **Warning**
>
> 这的确是有关IPv6支持不成熟问题的解决方法， 这在AIX 5.3版本发布期间有明显改善。它在AIX 5.3版本进行工作，但并不代表优雅的解决问题的办法。 报道称这种解决方法不仅是不必要的，而且会在AIX 6.1上导致问题， 其中IPv6的支持也日渐成熟。

15.7.1.4. 内存管理

AIX关于内存管理方式上有些特别。 你可以有很多倍的千兆字节RAM免费的服务器， 但是当运行应用程序时，仍然有内存或地址空间不足的错误。 `createlang` 导致不寻常错误的例子。 比如，作为PostgreSQL安装的所有者运行：

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql7
```

作为PostgreSQL安装的非拥有者进行运行：

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql7
```

另一个例子是在PostgreSQL服务器日志中内存不足的错误， 与每个内存分配接近或者大于256 MB。

所有这些问题的总体原因是默认bittedness并且使用服务器进程的内存模型。 默认情况下，所有的在AIX上编译的二进制文件是32位。 这不依赖于使用的硬件类型或内核。 这些32位进程是使用一些模型之一将4 GB内存限制为256 MB的段大小。 默认允许堆内小于256 MB，因为它共享单独的段与堆栈。

在支持 `createlang` 的情况下， 上面检查你的PostgreSQL安装中的umask和二进制文件的权限。 参与该示例中的二进制文件分别为32位和安装模式是750而不是755。 因为权限被以这种方式设置， 只有进程组中的所有者或成员才可以加载库。 因为它不是被所有人可读的， 加载器将对象置入过程堆而不是共享库段，它放置在那里。

此问题的"理想"解决方案是使用64位PostgreSQL编译， 但事实并非总是可行的，因为32位处理器系统可以进行， 但不能运行64位二进制文件。

如果需要32位二进制， 设置 `LDR_CNTRL` 到 `MAXDATA=0x``_n_` 0000000， 其中1 <= n <= 8，启动PostgreSQL服务器之前，尝试不同的值并且设置 `postgresql.conf` 以找到合理运行的配置。 `LDR_CNTRL` 的使用告诉AIX你想要服务器有 `MAXDATA` 字节空闲给堆，分配256 MB的段。

当你找到一个可行的配置， `ldedit` 可以用来修改二进制文件， 他们默认使用所需的堆大小。PostgreSQL可以也被重新编译， 通过配置 `configure LDFLAGS="-Wl,-bmaxdata:0x``_n_` 0000000"达到同样的效果。

对于64位版本，设置 `OBJECT_MODE` 为64 并且传递 `CC="gcc -maix64"` 和 `LDFLAGS="-Wl,-bbigtoc"` 到 `configure` 。 （ `xlc` 选项可能有所不同。） 如果省略 `OBJECT_MODE` 的输出， 您可能产生链接器错误。当设置 `OBJECT_MODE` 的时候， 它告诉AIX编译工具如 `ar` ， `as` 和 `ld` 默认处理对象的类型。

默认情况下，可能发生分页空间过量使用。 虽然我们没见过这种情况发生，当它用完内存并且产生过量访问时，AIX将杀死进程， 最接近这一点的是我们所看到的是交叉失败，因为系统决定没有足够的内存用于另一个进程。 像许多其他的AIX部分，如果这产生问题，那么 分页空间分配方法及内存不足杀进程在系统或进程范围基础上是可配置。

参考文献和资源

"Large Program Support", *AIX Documentation: General Programming Concepts: Writing and Debugging Programs*.

"Program Address Space Overview", *AIX Documentation: General Programming Concepts: Writing and Debugging Programs*.

"Performance Overview of the Virtual Memory Manager (VMM)", *AIX Documentation: Performance Management Guide*.

"Page Space Allocation", *AIX Documentation: Performance Management Guide*.

"Paging-space thresholds tuning", *AIX Documentation: Performance Management Guide*.

Developing and Porting C and C++ Applications on AIX, IBM Redbook.

15.7.2. Cygwin

PostgreSQL可以使用Cygwin，Windows的类Linux环境进行编译， 但该方法不如本地Windows编译(参阅Chapter 16)， 并且不再推荐在Cygwin下运行服务器。

当从源码进行编译时，按照正常的安装程序进行（也就是 `./configure;make` ）， 注意下面的Cygwin特殊差异：

- 在Windows功能之前使用Cygwin bin目录设置你的路径。这有助于避免编译问题。
- GNU make命令称为 `make` 而不是 `gmake` 。
- 不支持 `adduser` 命令；在Windows NT,2000或者XP上使用适当的用户管理应用。否则，忽略这一步。

- 不支持 `su` 命令；在Windows NT,2000或者XP上使用ssh模拟su。否则，忽略这一步。
- 不支持OpenSSL。
- 为了共享内存支持开启 `cygserver` 。 要做到这一点，输入命令 `/usr/sbin/cygserver&` 。这个程序需要运行于任何时候， 你启动PostgreSQL服务器或初始化一个数据库集群（ `initdb` ）。 可能需要改变默认 `cygserver` 配置（例如，增加 `SEMMNS` ）以避免 由于缺乏系统资源而导致的PostgreSQL失败。
- 在某些系统上编译可能会失败，它使用的语言环境不是C。 为了解决这个问题，在编译之前通过执行 `export LANG=C.utf8` 设置区域到C， 并且，你已经安装完PostgreSQL之后，设置它返回到以前的设置。
- 平行回归测试（ `make check` ）可以产生伪回归测试失败， 由于 `listen()` 积压队列溢出，这会导致连接拒绝错误或挂起。 你可以使用make变量 `MAX_CONNECTIONS` 限制连接数，因此：

  ```
  make MAX_CONNECTIONS=5 check
  ```

  在某些系统上你最多可以有10个并发连接。

可能安装 `cygserver` 和作为Windows NT服务的PostgreSQL服务器。 关于如何做的信息，请参阅包含Cygwin上的PostgreSQL二进制包的 `README` 文档。 它被安装在目录 `/usr/share/doc/Cygwin` 中。

15.7.3. HP-UX

PostgreSQL 7.3+应该在运行HP-UX 10.X或者11.X的系列700/800 PA-RISC机器上工作， 给予相应的系统补丁级别和编译工具。 至少有一个开发者经常在HP-UX 10.20上测试。 并且，我们在HP-UX 11.00和11.11上有成功安装的报道。

除了PostgreSQL源代码发布外，您将需要GNU make（HP make不能执行）， 以及GCC或HP的完整ANSI C 编译器。 如果你打算从Git 源代码而不是发布包编译，你还需要Flex（GNU lex）和Bison (GNU yacc)。我们建议确保你是最新的HP补丁。至少，如果你正在HP-UX 11.11上编译64位二进制文件，您可能需要PHSS_30966 (11.11)或 者后继补丁，否则 `initdb` 可能会挂起：

PHSS_30966 s700_800 ld(1) and linker tools cumulative patch

一般原则，你应该使用当前的libc和ld/dld补丁程序，以及编译器补丁， 如果你正在使用HP C编译器。 请参阅HP支持网站，比如http://itrc.hp.com和 ftp://us-ffs.external.hp.com/免费的最新补丁副本。

如果你正在PA-RISC 2.0的机器上编译，并希望 使用GCC的64位二进制文件，你必须使用GCC的64位版本。HP-UX PA-RISC的GCC二进制和Itanium可以从 http://www.hp.com/go/gcc 获得。不要忘了在同一时间安装binutils。

如果你想在PA-RISC 2.0机器上编译，并且在PA-RISC 1.1机器上编译二进制，你需要声明 `CFLAGS` 中的 `+DAportable` 。

如果你在HP-UX Itanium机器上进行编译，你将需要带有依赖包或者继承补丁的最新HP ANSI C编译器。

PHSS_30848 s700_800 HP C Compiler (A.05.57) PHSS_30849 s700_800 u2comp/be/plugin library Patch

如果你有HP的C编译器和GCC，那么当你运行 `configure` 时，可能想要明确选择使用的编译器。

```
./configure CC=cc
```

对于HP的C编译器，或者

```
./configure CC=gcc
```

GCC。如果你忽略这些设置，那么如果它有选择的话，configure将选择 `gcc` 。

缺省安装目标位置是 `/usr/local/pgsql` ，你可能想要在 `/opt` 下改变一些内容。 如果这样，使用 `--prefix` 切换 `configure` 。

在回归测试中，可能有一些几何测试中的低阶位数差异，这取决于您使用的编译器和数学库版本的不同而不同。 任何其他错误都是令人怀疑的。

15.7.4. IRIX

PostgreSQL已在MIPS r8000，r10000 (ip25和ip27)以及r12000(ip35)处理器成功运行， 运行在IRIX 6.5.5m, 6.5.12, 6.5.13和6.5.26与MIPSPro编译器版本7.30, 7.3.1.2m, 7.3和7.4.4m。

您将需要MIPSPro完整ANSI C编译器。尝试与GCC编译存在问题。 这是关于使用函数返回某种结构的众所周知的GCC bug（不固定为3.0版本）。 此错误会影响功能，如 `inet_ntoa` ， `inet_lnaof` ， `inet_netof` ， `inet_makeaddr` ，和 `semctl` 。它被认为是固定的，迫使代码链接使用libgcc的那些函数，但是这并没有被测试过。

MIPSPro编译器的7.4.1m版本产生错误代码是已知的。当尝试启动数据库时，现象是"无效主节点记录"。 版本7.4.4m可行；中间版本的状态是不确定的。

有可能是编译问题，如下：

```
cc-1020 cc: ERROR File = pqcomm.c, Line = 427
  The identifier "TCP_NODELAY" is undefined.

                  if (setsockopt(port->sock, IPPROTO_TCP, TCP_NODELAY,
```

一些版本包括在 `sys/xti.h` 中的TCP定义， 因此在 `src/backend/libpq/pqcomm.c` 和 `src/interfaces/libpq/fe-connect.c` 中添加 `#include <sys/xti.h>` 是必要的。如果你遇到这类问题，请让我们知道，以便于我们可以制定合适的修复。

在回归测试中，可能有一些几何测试中的低阶位数的差异，依赖于你使用的FPU。 任何其他的错误都值得怀疑。

15.7.5. MinGW/Native Windows

Windows下的PostgreSQL可以使用MinGW，微软操作系统的类Unix编译环境，或者使用 微软的Visual C++编译器套件编译。 MinGW的编译版本采用本章中描述的正常编译系统; Visual C++编译工作完全不同，并且在Chapter 16中描述。 它完全是一种本地编译，并且没有使用像MinGW的额外软件。 现成的安装程序可以从PostgreSQL的网站上获得。

本地Windows端口需要Windows 2000或更高的32或64位版本。早期操作系统 没有足够的基础设施（但Cygwin可以使用这些信息）。 MinGW，类Unix编译工具，以及MSYS，Unix工具集合 需要运行类似于 `configure` 的shell脚本， 可以从http://www.mingw.org/下载。 两者都不需要运行生成的二进制文件; 他们只需要创建二进制文件。

要使用MinGW编译64位二进制文件，安装来自http://mingw-w64.sourceforge.net/的64位工具集， 将其bin目录放在 `PATH` 中， 并运行 `--host=x86_64-w64-mingw` 选项的 `configure` 命令。

您已经安装了一切之后，建议您在 `CMD.EXE` 下运行psql， 作为MSYS控制台有缓冲问题。

15.7.5.1. Windows上搜索崩溃转储

如果PostgreSQL在Windows崩溃，有可能产生minidumps可用于跟踪崩溃的原因， 类似于在Unix上的核心转储。这些转储可以使用Windows调试器工具或者Visual Studio读取。 为了启动Windows上的转储， 创建名为集群数据目录里的 `crashdumps` 的子目录。 转储将被写入到该目录，连同基于崩溃进程识别符和崩溃当前时间的唯一名称。

15.7.6. SCO OpenServer和SCO UnixWare

PostgreSQL可以在SCO UnixWare 7和SCO OpenServer 5上编译。在OpenServer上， 你可以使用OpenServer Development Kit或者Universal Development Kit。 然而，可能需要一些调整，正如下面描述的。

15.7.6.1. Skunkware

你应该找到你的SCO Skunkware CD的副本。 Skunkware CD附带UnixWare 7和OpenServer 5当前版本。 Skunkware包括可从互联网获得的许多流行程序的准备安装版本。 例如，gzip, gunzip, GNU Make, Flex和Bison。 对于UnixWare 7.1，这个光盘现在标有"开放式许可软件补充"，如果你没有这个光盘， 可以从http://www.sco.com/skunkware/获得。

Skunkware有UnixWare和OpenServer的不同版本。 请确保您安装了您的操作系统的正确版本，除非另有说明如下。

在UnixWare 7.1.3及以上，包含在UDK CD上的GCC编译器作为GNU Make。

15.7.6.2. GNU Make

您需要使用GNU make程序，位于Skunkware CD。默认情况下，它作为 `/usr/local/bin/make` 安装。 为了避免混淆SCO `make` 程序，你可能需要重命名GNU `make` 为 `gmake` 。

由于UnixWare 7.1.3及以上，GNU Make程序是UDK CD的OSTK部分， 并且位于 `/usr/gnu/bin/gmake` 。

15.7.6.3. Readline

Readline库在Skunkware CD上。但它不包括在UnixWare 7.1 Skunkware CD上。如果你有UnixWare 7.0.0或者7.0.1 Skunkware CD，您可以从那里安装。 否则，尝试http://www.sco.com/skunkware/。

缺省情况下，Readline安装在 `/usr/local/lib` 和 `/usr/local/include` 。 然而，没有帮助的情况下PostgreSQL `configure` 程序不能找到， 如果你安装Readline,那么使用 `configure` 的下列选项：

```
./configure --with-libraries=/usr/local/lib --with-includes=/usr/local/include
```

15.7.6.4. 在OpenServer使用UDK

如果你正在OpenServer上使用新的通用开发Kit (UDK)编译器，你需要声明UDK库的位置：

```
./configure --with-libraries=/udk/usr/lib --with-includes=/udk/usr/include
```

将上面这些与Readline选项放在一起：

```
./configure --with-libraries="/udk/usr/lib /usr/local/lib" --with-includes="/udk/usr/incl
```

15.7.6.5. 阅读PostgreSQL手册页

缺省情况下，PostgreSQL手册页安装在 `/usr/local/pgsql/man` 。 缺省UnixWare不能查阅手册页，为了可用查看你需要修改 `/etc/default/man` 中的 `MANPATH` 变量，比如：

```
MANPATH=/usr/lib/scohelp/%L/man:/usr/dt/man:/usr/man:/usr/share/man:scohelp:/usr/local/ma
```

在OpenServer上，需要投入一些额外研究形成可用手册页，因为它不同于其他的平台。 目前，PostgreSQL根本不安装。

15.7.6.6. 7.1.1b功能补充的C99问题

对于先于发布OpenUnix 8.0.0（UnixWare 7.1.2)的编译器，包括7.1.1b功能补充， 你可能需要在 `CFLAGS` 或者 `CC` 环境变量中指定 `-Xb` 。 这个指示是编译 `tuplesort.c` 中引用内联功能的一个错误。 显然，有7.1.2（8.0.0）编译器及以上的变化。

15.7.6.7. UnixWare上的线程

对于线程，你必须在所有使用libpq的程序上使用 `-Kpthread` 。 libpq使用 `pthread_*` 调用， 其中只有 `-Kpthread` / `-Kthread` 标志可用。

15.7.7. Solaris

PostgreSQL在Solaris上有很好的支持。更新操作系统越多，您将遇到越少的问题; 详细信息请参见下面。

15.7.7.1. 所需工具

您可以使用GCC或Sun的编译器套件进行编译。为更好的代码优化， 在SPARC架构上强烈推荐Sun的编译器。当推荐使用GCC 2.95.1; GCC 2.95.3或更高版本的时候，我们已经听到问题报告。 如果您正在使用Sun的编译器，请注意不要选择 `/usr/ucb/cc` ;而使用 `/opt/SUNWspro/bin/cc` 。

您可以从http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/下载Sun Studio。 许多GNU工具都集成到Solaris 10上，或者它们目前在Solaris companion CD上。 如果你喜欢Solaris旧版本的包，你可以从http://www.sunfreeware.com中找到这些工具。 如果你喜欢源码， 可以参阅http://www.gnu.org/order/ftp.html。

15.7.7.2. OpenSSL问题

当您使用OpenSSL编译PostgreSQL的时候，你可能会得到以下文件中的编译错误：

- `src/backend/libpq/crypt.c`
- `src/backend/libpq/password.c`
- `src/interfaces/libpq/fe-auth.c`
- `src/interfaces/libpq/fe-connect.c`

这是因为在标准的 `/usr/include/crypt.h` 头部以及OpenSSL所提供的头文件之间有命名空间冲突。

升级您的OpenSSL安装到版本0.9.6a修复了这个问题。Solaris 9和上面有OpenSSL的较新版本。

15.7.7.3. configure抱怨失败测试程序

如果 `configure` 抱怨失败的测试程序， 这可能是运行时链接程序无法找到一些库的情况， 可能libz, libreadline或一些其他非标准库如libssl。 为了指向正确的位置，在 `configure` 命令行上设置 `LDFLAGS` 环境变量。 比如：

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

参阅ld手册页获取更多详情。

15.7.7.4. 64位编译有时崩溃

在Solaris 7以及更高版本，64位版本的libc中有一个 `vsnprintf` 程序， 从而导致PostgreSQL中不稳定的核心转储。 最简单的已知的解决方法是强制PostgreSQL使用 `vsnprintf` 自己的版本，而不是库副本。 要做到这一点，运行 `configure` 之后 编辑 `configure` 产生的文件： `src/Makefile.global` 中， 更改行

```
LIBOBJS =
```

读取

```
LIBOBJS = snprintf.o
```

（可能还有其他在这个变量中已经列出的文件。顺序无关紧要。）然后编译照常。

15.7.7.5. 编译以获得最佳性能

SPARC架构上，强烈推荐Sun Studio编译。尝试使用 `-xO5` 优化 标志产生显著快速的二进制文件。不要使用 任何的标志修改浮点运算的行为以及 `errno` 处理（例如， `-fast` ）。这些标志可以提高一些日期/时间计算中的PostgreSQL不规范行为。

如果您没有理由在SPARC上使用64位二进制文件，更喜欢32位版本。64位运算速度较慢，并且64位二进制比32位更慢。另一方面，在AMD64 CPU族中的32位代码不是本地的， 这就是为什么32位代码在这个CPU族中显著慢的原因。

15.7.7.6. 使用DTrace追踪PostgreSQL

是的，有可能使用DTrace。参阅 Section 27.4获取更多信息。 你也可以通过文章 https://blogs.oracle.com/robertlor/entry/user_level_dtrace_probes_in 查找更多的信息。

如果您看到 `postgres` 执行中断与错误信息的连接，如：

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
gmake: *** [postgres] Error 1
```

您的DTrace安装太旧而不能处理静态函数的探测。你需要Solaris 10u4或更高版本。

Chapter 16. Windows下用源代码安装

Table of Contents

对于需要在windows下安装PostgreSQL的大多数普通用户来说， 推荐从官网下载图形化界面的二进制安装包。源代码安装主要面向PostgreSQL开发人员及相关扩展插件的开发人员。

在Windows中,有多种方法编译安装PostgreSQL。 对于微软工具的话，最简单的方法是安装一个Visual Studio Express 2012 for Windows Desktop，并使用它自带的编译器进行编译。也可以使用Microsoft Visual C++ 2005, 2008 or 2010来编译安装。 在一些情况下，除了编译器还需要安装Windows SDK。

此外，可以使用MinGW提供的GNU编译工具来编译PostgreSQL。 如果Windows系统版本比较旧， 可以使用Cygwin进行编译安装。

最后，为了兼容静态链接(libpq)的应用， 可以用Visual C++ 7.1或Borland C++ 来编译libpq。

使用MinGW 或者Cygwin的普通编译系统的话， 可以参看Chapter 15及Section 15.7.5和Section 15.7.2。 要在这些环境中生成原生的64位可执行程序，可以使用MinGW-w64工具。这些工具也可以在其它的平台下执行交叉编译，生成32位或64位的Windows可执行程序， 例如Linux 和Darwin系统。 在生产环境中不推荐使用Cygwin， 它仅适合用于Windows 98等比较旧的Windows版本上的编译器编译不了时使用。 官方的的二进制可执行文件是由Visual Studio编译的。

原生的psql可执行程序不支持命令行编辑。 而Cygwin编译的可执行文件支持命令行编辑， 因此需要在Windows下交互式使用psql的话， 应该使用这种方式编译。

16.1. 用Visual C++或Microsoft Windows SDK编译

PostgreSQL可以使用微软的Visual C++编译器套件来编译。 这些编译器可以是Visual Studio, Visual Studio Express或者某些版本的Microsoft Windows SDK。 如果没有安装Visual Studio的环境， 最简单的方法是使用Windows SDK 7.1或者Visual Studio Express 2012 for Windows Desktop中的编译器, 它们都可以从微软官网免费下载。

PostgreSQL支持从Visual Studio 2005到Visual Studio 2012(包括精简版)的编译器, 以及6.0到7.1版本的独立的Windows SDK编译器。 64位的PostgreSQL只支持6.0a到7.1版本的Microsoft Windows SDK和Visual Studio 2008及其以上版本。

用Visual C++或Platform SDK编译的工具在 `src/tools/msvc` 目录下。 编译时，确保系统路径下没有包含MinGW或Cygwin的工具，同时所需要的Visual C++工具在系统路径下是可用的。 在Visual Studio中，启动Visual Studio Command Prompt。 如果需要编译64位的版本，使用64位版本的命令，反之亦然。在Microsoft Windows SDK中， 从开始菜单中SDK下的列表启动CMD shell。在最新版的SDK中， 你可以使用 `setenv` 等命令来更改目标的CPU架构，编译类型和操作系统类型。 指令 `setenv /x86 /release /xp` 用来编译Windows XP或更新的32位版本。 使用 `/?` 命令可以查看 `setenv` 的其它用法。 所有命令都应该在 `src\tools\msvc` 目录下运行。

在编译前，根据需要修改 `config.pl` 中相应选项和使用到的第三方库的路径。 完整的配置由首先读取和解析的 `config_default.pl` 决定，然后从 `config.pl` 获取变更。 例如，为了指定Python的安装路径，在 `config.pl` 文件中添加以下语句:

```
$config->{python} = 'c:\python26';
```

只需要指定和 `config_default.pl` 文件中有差别的参数。

如果需要指定其它的环境变量，创建一个 `buildenv.pl` 文件并写入所需要的命令。 例如，为了在PATH中添加一个还不存在的bison变量，创建一个文件包含以下内容：

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

16.1.1. 要求

编译PostgreSQL还需要以下几个工具。 使用 `config.pl` 文件来指定这些库的路径。

Microsoft Windows SDK

如果编译环境不兼容所支持的Microsoft Windows SDK版本， 建议把SDK的版本升级到最新的版本(目前为7.1版本),下载地址为http://www.microsoft.com/downloads/。

在SDK中，必须要保证包含Windows Headers and Libraries这部分内容。 如果安装的Windows SDK中已经包含了Visual C++ Compilers， 就不再需要使用Visual Studio进行编译了。 注意Windows SDK 8.0a之后的版本不再附有完整的命令行编译环境。

ActiveState Perl

ActiveState Perl用于编译生成脚本，而MinGW或Cygwin的Perl则不起作用。 同样，它的路径也要添加到系统路径中。二进制文件下载地址为http://www.activestate.com (注意: 必须使用5.8或者更新的版本， 免费的标准版既可）。

下面的工具在开始时是不需要的，但是对于完整的包的编译是必须的。 使用 `config.pl` 文件来指定这些库的路径。

ActiveState TCL

用于编译PL/TCL(注意:要求8.4版本， 免费的标准版既可）。

Bison and Flex

Bison and Flex用来从GIT进行编译，对于发行版文件的编译是不需要的。 Bison需要1.875或2.2及更新版本。Flex需要使用2.5.31或更新版本。

Bison and Flex包含在msys工具套件内。 作为MinGW编译器套件的一部分，msys可以从http://www.mingw.org/wiki/MSYS下载。 同时作为msysGit的一部分，msys也可以从http://git-scm.com/下载。

在 `buildenv.pl` 中，需要把包含 `flex.exe` 和 `bison.exe` 的目录添加到系统路径中， 除非它们已经存在于系统路径中。对于MinGW, 要把MinGW安装目录的子目录路径 `\msys\1.0\bin` 添加到系统路径中。 对于msysGit,要把Git安装目录下的 `bin` 目录添加到系统路径中。不要添加MinGW编译工具本身的目录到系统路径中。

> **Note:** GnuWin32中的Bison存在一个bug，当文件安装的路径中含有空格的时候，会导致Bison出现故障。 例如，用英文安装时的默认路径 `C:\Program Files\GnuWin32` 时会出现上述问题，解决办法是安装在类似 `C:\GnuWin32` 这样的目录， 或者在GnuWin32的环境变量PATH中使用NTFS短名称路径(例如 `C:\PROGRA~1\GnuWin32`)。
>
> **Note:** 在PostgreSQL FTP上的winflex是旧的二进制版本， 在64位Windows主机上执行时， 会报"flex: fatal internal error, exec failed"的错误。可以使用msys中的flex来替代以避免这个问题。

Diff

Diff用于进行回归测试，可以从http://gnuwin32.sourceforge.net下载。

Gettext

Gettext用于编译NLS支持，可以从http://gnuwin32.sourceforge.net下载。 需要注意的是二进制，依赖项和开发文件都需要。

MIT Kerberos

MIT Kerberos 用于提供Kerberos认证支持，可以从http://web.mit.edu/Kerberos/dist/index.html下载。

libxml2 and libxslt

libxml2和libxslt用于提供XML支持。可以从http://zlatkovic.com/pub/libxml下载二进制文件，也可以从http://xmlsoft.org下载源代码。需要注意，libxml2需要iconv，也在上面的地址下载。

openssl

openssl用于提供SSL支持。可以从http://www.slproweb.com/products/Win32OpenSSL.html下载二进制文件， 或者在http://www.openssl.org下载源代码。

ossp-uuid

ossp-uuid用于提供UUID-OSSP支持(仅用于contrib)。可以从http://www.ossp.org/pkg/lib/uuid/下载。

Python

Python用于编译PL/Python。可以从http://www.python.org下载。

zlib

zlib用于为pg_dump和pg_restore提供压缩支持。可以从http://www.zlib.net下载二进制文件。

16.1.2. 针对64位Windows的注意事项

PostgreSQL的X64架构只能在64位Windows系统上编译，不支持安腾处理器。

同一个编译树上不支持32位和64位的混编。编译系统能自动检测平台环境是32位或64位， 并编译出对应版本。因此，在编译前启动正确的命令行很重要。

在服务器端使用python或openssl这样的第三方库，必须是64位的。64位服务器上不支持对32位库的加载。 一些第三方库只支持32位的PostgrSQL，这种情形下，它们不能用在64位的PostgreSQL上。

16.1.3. 编译

使用发行版配置(默认)来编译完整的PostgreSQL，运行命令：

```
<kbd class="literal">build</kbd>
```

使用调试配置来编译完整的PostgreSQL，运行命令：

```
<kbd class="literal">build DEBUG</kbd>
```

只编译单个项目，例如psql，运行命令：

```
<kbd class="literal">build psql</kbd>
<kbd class="literal">build DEBUG psql</kbd>
```

要将默认的编译配置改为调试配置，需要在 `buildenv.pl` 文件中加入下面的内容：

```
$ENV{CONFIG}="Debug";
```

也可以Visual Studio的图形化界面来编译，这时需要在命令行中运行：

```
<kbd class="literal">perl mkvcbuild.pl</kbd>
```

然后在Visual Studio中打开 `pgsql.sln` 文件(从资源树的根目录)。

16.1.4. 清除和安装

大多数时候，Visual Studio会自动追踪所依赖的文件的改动。但是当有大量改动的的时候，可能需要清除安装。 要做到这一点，只需简单的运行 `clean.bat` ，它将自动的清除生成的相关文件。 也可以在运行时添加 `dist` 参数，它的行为和<kbd class="literal">make distclean</kbd>命令比较相似，同时会清除flex/bison的输出文件。

默认情况下，所有的文件都会写在 `debug` 或 `release` 目录的一个子目录下面。要使用标准设置安装这些文件， 初始化这些生成的文件并使用数据库，运行命令：

```
<kbd class="literal">install c:\destination\directory</kbd>
```

16.1.5. 运行回归测试

要运行回归测试，首先确保已经完成所需的编译。同时，确保用来加载系统所有部分(例如程序语言Perl和Python的DLLs)的DLLs已经在系统路径中存在。 如果没有，通过 `buildenv.pl` 文件来设置。要运行测试，运行 `src\tools\msvc` 目录中的一个命令：

```
<kbd class="literal">vcregress check</kbd>
<kbd class="literal">vcregress installcheck</kbd>
<kbd class="literal">vcregress plcheck</kbd>
<kbd class="literal">vcregress contribcheck</kbd>
```

要改变调度方式(默认是并行),将它加在命令行后：

```
<kbd class="literal">vcregress check serial</kbd>
```

关于回归测试的更多内容，见Chapter 30。

16.1.6. 编译文档

把PostgreSQL文档编译成HTML格式的需要一些工具和文件。为所有这些文件创建一个根目录，将它们存储在下列的子目录中。

OpenJade 1.3.1-2

从 http://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/download下载并解压到子目录 `openjade-1.3.1` 中。

DocBook DTD 4.2

从http://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip下载并解压到子目录 `docbook` 中。

DocBook DSSSL 1.79

从http://sourceforge.net/projects/docbook/files/docbook-dsssl/1.79/docbook-dsssl-1.79.zip/download下载并解压到子目录 `docbook-dsssl-1.79` 中。

ISO character entities

从http://www.oasis-open.org/cover/ISOEnts.zip下载并解压到子目录 `docbook` 中。

编辑 `buildenv.pl` 文件，为根目录的地址增加一个变量，如：

```
$ENV{DOCROOT}='c:\docbook';
```

要编译文档，运行 `builddoc.bat` 文件。注意为了生成索引，实际上编译运行来两次。生成的HTML文件存在于 `doc\src\sgml` 中。

16.2. 用Visual C++或 Borland C++编译 libpq

只有在需要一个含有不同的调试和发行版标记的版本，或者需要一个静态库来链接应用程序时， 才推荐使用Visual C++ 7.1-9.0或Borland C++来编译libpq。 一般情况下，推荐使用MinGW或Visual Studio或Windows SDK的方法。

采用Visual Studio 7.1 or later编译libpq的客户端库，使用下面的命令修改`src`的路径：

```
<kbd class="literal">nmake /f win32.mak</kbd>
```

采用Visual Studio 8.0 or later编译64位的libpq客户端库，使用下面的命令修改`src`的路径：

```
<kbd class="literal">nmake /f win32.mak CPU=AMD64</kbd>
```

关于支持的变量的更多细节参考`win32.mak`文件。

采用Borland C++编译libpq的客户端库，使用下面的命令修改`src`的路径：

```
<kbd class="literal">make -N -DCFG=Release /f bcc32.mak</kbd>
```

16.2.1. 生成文件

下面的文件会被编译：

`interfaces\libpq\Release\libpq.dll`

可连接的前端动态库

`interfaces\libpq\Release\libpqdll.lib`

导入库来连接程序和`libpq.dll`

`interfaces\libpq\Release\libpq.lib`

静态版的前端库

通常不需要安装任何的客户端文件。应该将`libpq.dll`文件放在和可执行的应用程序文件放在同一个目录下。 除非必要，否则不要把`libpq.dll`放入`Windows`，`System`或`System32`的目录下。 如果一个文件是使用安装程序安装的，那么需要使用`VERSIONINFO`进行版本审查，以免被新版本的库覆盖掉。

如果需要在本机器上使用libpq进行开发，要将`src\include`和`src\interfaces\libpq`的子目录加入到编译器配置的资源树中。

要使用一个库，必须将 `libpqdll.lib` 添加到项目中。(在Visual C++,只需在项目上右键单击并选择添加。)

Chapter 17. 服务器设置和操作

Table of Contents

本章讨论如何设置和运行数据库服务器以及它如何与操作系统交互。

17.1. PostgreSQL用户账户

与任何可以从外界访问的服务器守护进程一样，我们也建议用一个独立的用户帐户运行PostgreSQL。这个用户帐户应该拥有由这个服务器管理的数据， 而且不应该与其它守护进程共享这些数据。比如，用 `nobody` 用户是个烂主意。 我们不建议把可执行文件安装为由此用户所有，因为这样一来被攻破的系统就可以修改它们自己拥有的二进制文件。

要向系统里增加用户帐户，参考 `useradd` 或 `adduser` 命令。 我们经常使用postgres的用户名，并且在本书中都假设这个名字， 但你不必这么做。

17.2. 创建数据库集群

在做任何事情之前，必须先初始化磁盘上的数据存储区，叫作数据库集群 (标准SQL术语称为"目录集群")。一个数据库集群是一系列数据库的集合， 这些数据库可以通过单个数据库服务器的实例管理。在初始化后，一个数据库集群将包含一个叫 `postgres` 的数据库，这个库是给工具、用户和第三方程序使用的缺省数据库。数据库服务器本身并不要求 `postgres` 数据库的存在，但是很多外部工具假设它存在。另外一个在每个集群初始化过程中创建的数据库叫 `template1` 。 正如其名一样，这个数据库将作为随后创建的数据库的模版；在实际工作中不应该使用这个库 (参阅Chapter 21获取有关创建数据库的信息)。

用文件系统的术语来说，一个数据库集群是一个目录，所有数据都将存放在这个目录中。 我们把它称做数据目录或数据区。 在哪里存放数据完全取决于你的选择，我们没有缺省值，尽管 `/usr/local/pgsql/data` 或 `/var/lib/pgsql/data` 这样的目录很常用。要初始化一个数据库集群， 可以使用initdb,命令， 这个命令与PostgreSQL一起安装。你可以用 `-D` 选项指定数据目录的位置，例如：

```
<samp class="literal">$</samp> <kbd class="literal">initdb -D /usr/local/pgsql/data</kbd>
```

你必须以PostgreSQL用户的身份来执行这条命令， 这一点我们在前面一节描述过。

> **Tip:** 作为 `-D` 选项的替代品，你还可以使用 `PGDATA` 环境变量。

可选的，你可以通过pg_ctl程序运行 `initdb`，像这样：

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl -D /usr/local/pgsql/data initd
```

如果你使用 `pg_ctl` 启动或停止服务器（参阅Section 17.3）可能会更直观， 所以 `pg_ctl` 将会是你管理数据库服务器实例唯一使用的命令。

如果你声明的路径还不存在， `initdb` 将试图创建它。如果你按照我们的建议创建了一个非特权帐户的话， 你很有可能缺少做这些事情的权限。这时，你可以自己创建该目录(以 root 身份)然后把该目录的所有权交给 PostgreSQL用户。下面是可能有效的方法：

```
root# <kbd class="literal">mkdir /usr/local/pgsql/data</kbd>
root# <kbd class="literal">chown postgres /usr/local/pgsql/data</kbd>
root# <kbd class="literal">su postgres</kbd>
postgres$ <kbd class="literal">initdb -D /usr/local/pgsql/data</kbd>
```

如果数据目录看起来像已经初始化过了，那么 `initdb` 会拒绝运行。

因为数据目录包含所有存储在数据库里的数据，所以出于安全考虑，这个目录不能给任何非授权用户访问。 因此， `initdb` 禁止除PostgreSQL用户帐户以外的任何用户访问这个目录。

不过，因为目录的内容是安全的，所以缺省的客户端认证设置允许任意本地用户连接到数据库甚至成为超级用户。 如果你不信任本地用户，我们建议你使用 `initdb` 的 `-W`， `--pwprompt` 或 `--pwfile` 选项给超级用户赋予一个口令。还有，声明 `-A md5` 或 `-A password`， 这样就不会使用缺省的 `trust` 身份认证。或者在执行 `initdb` 之后，第一次启动服务器之前 修改 `pg_hba.conf` 文件。另外一些合理的方法包括 `peer` 认证或者用文件系统权限禁止连接。参阅Chapter 19获取更多细节。

`initdb` 同时也为数据库集群初始化缺省区域。通常， 它将只是使用环境中的区域设置并且把它们应用于初始化的数据库。我们可以为数据库声明不同的区域； 有关这些的更多信息可以在Section 22.1中找到。在特定数据库集群里的缺省排序顺序是由 `initdb` 设置的，而且当你能用不同的排序顺序创建新的数据库时， initdb创建的模板数据库中使用的排序不能改变，除非删除并重新创建它们。使用非 `C` 或 `POSIX` 的区域还会有性能影响。因此，第一次就选择正确很重要。

`initdb` 还为数据库集群设置缺省的字符集编码。通常这个应该选择与区域匹配。 详见Section 22.3。

17.2.1. 网络文件系统

许多安装在网络文件系统创建数据库集群。有时直接通过NFS或通过使用网络附加存储(NAS)设计内部使用NFS。PostgreSQL并不为NFS文件系统做什么特别的， 意味着他假设NFS和本地连接驱动器(DAS,直接附加存储)的行为一样。如果客户端和服务器NFS 实现有非标准的语义，会引起可靠性问题（参阅http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html）。 特别的，延迟的（异步的）写入NFS服务器可能会引起可靠性问题；如果可能， 同步的（没有缓存）安装NFS文件系统以避免这个问题。同样，不建议软安装NFS。 （存储区域网络(SAN)使用一个低级的通信协议而不是NFS。）

17.3. 启动数据库服务器

在任何人可以访问数据库前，你必须启动数据库服务器。数据库服务器程序名叫 `postgres`，它必须知道在哪里能找到它要用的数据。 这是利用 `-D` 选项实现的。因此，启动服务器最简单的方法是像下面这样：

```
$ <kbd class="literal">postgres -D /usr/local/pgsql/data</kbd>
```

这样将把服务器放在前台运行。这个步骤同样必须以PostgreSQL用户帐户登录来做。 没有 `-D` 选项，服务器将使用环境变量 `PGDATA` 命名的目录； 如果这个环境变量也没有，将导致失败。

通常，最好在后台启动 `postgres`，使用下面的 Unix shell 语法：

```
$ <kbd class="literal">postgres -D /usr/local/pgsql/data >logfile 2>&1 &</kbd>
```

把服务器的stdout和stderr放到某个地方是非常重要的， 就像在上面建议的这样。这样做既可以帮助审计又可以帮助诊断问题。 参阅Section 23.3获取有关日志文件处理的更完整讨论。

`postgres` 还接受一些其它的一些命令行选项。 更多的信息请参考postgres手册页和下面的Chapter 18。

这些 shell 语法很容易让人觉得无聊。因此我们提供了封装程序 pg_ctl以简化一些任务。比如：

```
pg_ctl start -l logfile
```

将在后台启动服务器并且把输出放到指定的日志文件中。 `-D` 选项和你直接运行 `postgres` 时的意思是一样的。 `pg_ctl` 还可以用于关闭服务器。

通常，你会希望在计算机启动的时候启动数据库服务器。 自动启动脚本是与操作系统相关的。PostgreSQL自己带了几个， 放在 `contrib/start-scripts` 目录里。需要 root 权限安装它们。

不同的系统在引导时有不同的启动守护进程的方法，所以我们建议你先熟悉它。 许多系统有名字称为 `/etc/rc.local` 或 `/etc/rc.d/rc.local` 这样的文件，其它的还有 `rc.d` 目录。不管你怎么做，都要记住服务器必须以 PostgreSQL用户帐户，而不是 *root* 或者任何其它用户身份运行。因此，你可能总是要用 `su postgres -c '...'` 这样的命令。比如：

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

下面是一些比较详细的与操作系统相关的建议。请注意把每个例子里的具体数值替换成合适的安装路径和用户名。

- 对于FreeBSD，看看PostgreSQL 源代码版本里的 `contrib/start-scripts/freebsd` 文件。
- 在OpenBSD上，把下面几行加到 `/etc/rc.local` 文件里：

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
    su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D
    echo -n ' postgresql'
fi
```

- 在Linux系统里，要么往 `/etc/rc.d/rc.local` 或 `/etc/rc.local` 文件里加上下面几行：

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

  要么看看PostgreSQL源代码树里的 `contrib/start-scripts/linux` 文件。

- 在NetBSD上，你可以根据爱好选择FreeBSD 或Linux的启动脚本之一。
- 在Solaris上，创建一个叫 `/etc/init.d/postgresql` 的文件， 包含下面行：

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/da
```

  然后在 `/etc/rc3.d` 里创建一个指向它的符号链接，名字叫 `S99postgresql` 。

运行的时候，它的PID是保存在数据目录下的 `postmaster.pid` 文件里的。这样做是为了避免多个服务器在同一个数据目录内运行，此文件同样可以用于关闭服务器。

17.3.1. 服务器启动失败

有几个非常常见的原因会导致服务器启动失败。通过检查服务器日志或者使用手工启动的方法 (不做 stdout 和 stderr 的重定向)，就可以看到错误信息。下面我们更详细地解释了其中一些错误信息。

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and
FATAL:  could not create TCP/IP listen socket
```

就像它提示的那样：你试图在已经有一个服务器运行着的端口上再运行了一个服务器。不过， 如果内核的错误信息不是 <samp class="literal">Address already in use</samp> 或者是其它的变种， 那就有可能是别的毛病。比如，试图在一个保留的端口上运行服务器会收到下面这样的信息：

```
$ <kbd class="literal">postgres -p 666</kbd>
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and
FATAL:  could not create TCP/IP listen socket
```

像这样的信息：

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

可能意味着内核对共享内存区的限制小于PostgreSQL试图分配的缓冲区大小 (本例中是 4011376640 字节)。或者可能意味着你根本就没有配置 System-V 风格的共享内存支持。 作为一个临时的解决办法，你可以试着以小于正常数量的缓冲区数(shared_buffers) 启动服务器。你最终还是会希望重新配置内核，以增加共享内存的尺寸。 如果你试图在同一台机器上启动多个服务器，而且它们所需的总空间超过了内核的限制，也会报这个错。

像下面这样的错误：

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

并不意味着着你已经用光磁盘空间了。它的意思是内核的 System V信号灯的限制小于PostgreSQL想创建的数量。 和上面一样，你可以通过减少允许的连接数(max_connections)来绕开， 但最终你还是会希望修改内核的限制。

如果你收到一个"illegal system call"错误，那么很有可能是内核根本不支持共享内存或者信号灯。 如果是这样的话，你唯一的选择就是重新配置内核并且把这些特性打开。

关于配置系统System V IPC资源的细节见Section 17.4.1。

17.3.2. 客户端连接问题

尽管可能在客户端出现的错误条件范围宽广，而且还和应用相关，但的确有几种错误与服务器的启动方式直接相关。 除了下面提到的几种错误以外的问题都应该在相应的客户端应用的文档中。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

这是纯粹的"我找不到可以交谈的服务器"错误。当试图进行 TCP/IP 通讯时它看起来像上面的样子。 常见的错误是忘记把服务器配置成允许 TCP/IP 连接。

另外，当试图通过一个 Unix 套接字与本机服务器通讯时，你会看到这个：

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

最后一行可以有效地验证客户端进行连接尝试时是否连对了位置。如果实际上没有服务器在那里运行，典型的内核错误是像上面显示的那样或者是<samp class="literal">Connection refused</samp>或 <samp class="literal">No such file or directory</samp>。尤其要注意的是这种环境下 <samp class="literal">Connection refused</samp>的信息显示并不 意味着服务器收到连接然后拒绝了连接。那样的话会产生一个不同的信息(像 Section 19.4里面显示的那样)。其它像 <samp class="literal">Connection timed out</samp>这样的信息表示更基本的问题，比如缺少网络连接等。

17.4. 管理内核资源

PostgreSQL有时可能耗尽各种操作系统的资源上限，尤其是多个服务器副本在同一个系统上运行时， 或者在一个非常大的安装时。这种情况说明了PostgreSQL 使用的内核资源和解决问题可以采取的步骤都和内核资源消耗有关。

17.4.1. 共享内存和信号灯

共享内存和信号灯的正确叫法是"System V IPC" (还有消息队列，不过与PostgreSQL无关)。， PostgreSQL只在Windows上自己提供这套机制的替换实现， 要运行PostgreSQL这些机制是必需的。

完全缺少这些机制的表现通常是在服务器启动时的Illegal system call错误。 这时除了重新配置内核外别无选择。PostgreSQL没它们干不了活。 这种情况很少见，但是，在现代操作系统上会出现。

如果PostgreSQL超出了这些IPC资源的硬限制之一的时候就会拒绝启动， 并且留下一条相当有启发性的错误信息，描述问题以及需要为它做些什么 (又见Section 17.3.1)。相关的内核参数在不同系统之间有着相对固定的术语； Table 17-1是一个概况。不过，设置它们的方法却多种多样。 下面给出一些平台的建议。

> **Note:** PostgreSQL 9.3之前，System V共享内存的数量需要启动的服务器大得多。 如果你运行更老的服务器版本，请参考你的服务器版本的文档。

Table 17-1. System V IPC参数

名字	描述	合理取值
`SHMMAX`	最大共享内存段尺寸(字节)	至少 1kB (如果运行多个服务器副本需要更多)
`SHMMIN`	最小共享内存段尺寸(字节)	1
	可用共享内存	

`SHMALL`	数量(字节或者页面)	面， `ceil(SHMMAX/PAGE_SIZE)`
`SHMSEG`	每进程最大共享内存段数量	只需要 1 个段，不过缺省比这高得多。
`SHMMNI`	系统范围最大共享内存段数量	类似 `SHMSEG` 加上用于其它应用的空间
`SEMMNI`	信号灯标识符的最小数量(也就是套)	至少 `ceil((max_connections + autovacuum_max_workers + 4) / 16)`
`SEMMNS`	系统范围的最大信号灯数量	`ceil((max_connections + autovacuum_max_workers + 4) / 16) * 17` 加上用于其它应用的空间
`SEMMSL`	每套信号灯最小信号灯数量	至少 17
`SEMMAP`	信号灯映射里的记录数量	参阅本文
`SEMVMX`	信号灯的最大值	至少 1000， 缺省通常是 32767， 除非被迫，否则不要修改

PostgreSQL的每个服务器的副本需要System V共享内存的少许字节（在64为平台上典型为48字节）。 在大多数现在的操作系统上，可以很容易的分配数量。然而，如果你运行了服务器的多个副本， 或者其他应用也使用System V共享内存，那么增大 `SHMMAX` 可能是必要的，共享内存段或 `SHMALL` 的最大字节大小，为系统范围System V共享内存的总数量。注意 `SHMALL` 在许多系统上是用页面数而不是字节数来计算的。

不太可能出问题的是共享内存段的最小尺寸(`SHMMIN`)，对PostgreSQL 来说大约是32字节左右(通常只是 1)，而系统范围(`SHMMNI`)或每进程(`SHMSEG`) 最大共享内存段数量不应该会产生问题，除非你的系统把它们设成零。

PostgreSQL每个允许的连接使用一个信号灯(max_connections)， 并且允许autovacuum工作进程(autovacuum_max_workers)，以 16 个为一套。 每套信号灯还包含第 17 个信号灯，它里面存储一个"magic number"， 以检测和其它应用使用的信号灯集冲突。系统里的最大信号灯数目是由 `SEMMNS` 设置的， 因此这个值应该至少和 `max_connections` 加上 `autovacuum_max_workers` 设置一样大， 并且每 16 个连接和工作还要另外加一个(参阅Table 17-1里面的公式)。 参数 `SEMMNI` 决定系统里一次可以存在的信号灯集的数目。 因此这个参数至少应该为 `ceil((max_connections + autovacuum_max_workers + 4) / 16)` 。 降低允许的连接数目是一个临时的绕开失败的方法，这个启动失败通常被来自函数 `semget` 的错误响应"No space left on device"搞得很让人迷惑。

有时候还可能有必要增大 `SEMMAP` ，使之至少按照 `SEMMNS` 配置。 这个参数定义信号灯资源映射的尺寸，可用的每个连续的信号灯块在这个映射中存放一条记录。 每当一套信号灯被释放，那么它要么会加入到该映射中一条相连的已释放块的入口中， 要么注册成一条新的入口。如果映射填满了碎片，那么被释放的信号灯就丢失了(除非重启)。 因此信号灯空间的碎片时间长了会导致可用的信号灯比应该有的信号灯少。

`SEMMSL` 参数决定一套信号灯里可以有多少信号灯， 对于PostgreSQL而言应该至少是 17 。

许多设置与"semaphore undo"(信号灯恢复)有关，比如 `SEMMNU` 和 `SEMUME` ， 这些与PostgreSQL无关。

AIX

至少对于版本 5.1 而言，我们没有必要为类似 `SHMMAX` 这样的参数做特殊的配置， 因为这个参数可以配置为所有内容都当作共享内存使用。这就是类似DB/2 这样的数据库常用的配置。

不过，我们可能有必要在 `/etc/security/limits` 里面修改全局 `ulimit` 信息， 因为文件大小的缺省硬限制(`fsize`)以及文件数(`nofiles`)可能太低了。

FreeBSD

缺省设置可以用 `sysctl` 或 `loader` 接口来修改。 下面的参数可以用 `sysctl` 设置：

```
&lt;samp class="literal"&gt;#&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl kern.ipc.shm
&lt;samp class="literal"&gt;#&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl kern.ipc.shm
&lt;samp class="literal"&gt;#&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl kern.ipc.sem
```

要想让这些设置重启后有效，修改 `/etc/sysctl.conf` 文件。

如果用 `sysctl`，那么剩下的信号灯设置是只读的，但是可以在 `/boot/loader.conf` 里设置：

```
kern.ipc.semmni=256
kern.ipc.semmns=512
kern.ipc.semmnu=256
```

修改完这些值以后需要重启以使新的设置生效。

你可能还想配置内核，把共享内存锁到 RAM 里，避免他们被交换到交换分区中。 这些可以通过使用 `sysctl` 设置 `kern.ipc.shm_use_phys` 来完成。

如果通过启用sysctl的 `security.jail.sysvipc_allowed` 运行在 FreeBSD jail 中， 那么必须将postmaster以不同的用户身份运行在不同的 jail 中。这样有助于增强安全性， 因为它防止了非root 用户干扰不同 jail 中的共享内存或信号灯，并且允许 PostgreSQL IPC 清理代码功能。在 FreeBSD 6.0 及之后的版本中，IPC 清理代码并不能正确侦测在其它 jail 中的进程， 因此无法防止其它 jail 中的 postmaster 进程占用相同的端口。

FreeBSD 4.0 之前的版本类似OpenBSD(见下文)。

NetBSD

NetBSD 5.0及以后，IPC参数可以用 `sysctl` 调整，例如：

```
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl -w kern.ipc.
```

要想让这些设置重启后有效，修改 `/etc/sysctl.conf` 文件。

你可能还想配置内核，把共享内存锁到 RAM 里，避免他们被交换到交换分区中。 这些可以通过使用 `sysctl` 设置 `kern.ipc.shm_use_phys` 来完成。

NetBSD 4.0 之前的版本类似OpenBSD(见下文)， 除了参数应该用关键字 `options` 而不是 `option` 来设置。

OpenBSD

编译内核时需要把选项 `SYSVSHM` 和 `SYSVSEM` 打开(缺省是打开的)。 共享内存的最大尺寸是由选项 `SHMMAXPGS` (以页计)决定的。 下面显示了一个如何设置这些参数的例子：

```
option        SYSVSHM
option        SHMMAXPGS=4096
option        SHMSEG=256

option        SYSVSEM
option        SEMMNI=256
option        SEMMNS=512
option        SEMMNU=256
option        SEMMAP=256
```

你可能还想配置内核，把共享内存锁在 RAM 中以避免它们被交换出去， 我们可以通过使用 `sysctl` 设置 `kern.ipc.shm_use_phys` 来完成。

HP-UX

缺省设置看来对普通安装是足够的了。对于HP-UX 10， `SEMMNS` 的出厂缺省是 128， 可能对大的数据库节点来说太小了。

IPC可以在System Administration Manager(SAM)下面的 Kernel Configuration->Configurable Parameters配置。 配置完了以后选择Create A New Kernel选项。

Linux

缺省最大段大小为32MB，缺省的最大总字节为2097152页。一页通常是4096字节， 除了带有"huge pages"的不寻常的内核配置（使用 `getconf PAGE_SIZE` 来校验）。

共享内存大小设置可以通过 `sysctl` 接口改变。例如，要允许16 GB:

```
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl -w kernel.sh
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;sysctl -w kernel.sh
```

为了这些设置在重启后保持有效，将这些设置放到 `/etc/sysctl.conf` 里。这样做是高度推荐的。

老版本里可能没有 `sysctl` 程序，但是同样的改变可以通过操作 `/proc` 文件系统来做：

```
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;echo 17179869184 &g
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;echo 4194304 &gt;/p
```

剩下的缺省是相当宽松的大小，通常不需要改变。

Mac OS X

在 OS X 中配置共享内存推荐的方法是创建一个名为 `/etc/sysctl.conf` 的文件， 包含变量赋值，例如：

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

注意在某些OS X版本里，所有五个共享内存参数必须都在 `/etc/sysctl.conf` 中设置，否则将会被忽略。

还要注意最近版本的 OS X 将拒绝把 `SHMMAX` 的数值设置为非 4096 的倍数。

在这个平台上， `SHMALL` 是用 4KB 页来度量的。

在老的OS X版本里，你将需要重启以使共享内存配置的改变生效。自OS X 10.5起，在运行中修改除了 `SHMMNI` 的所有参数成为可能，使用sysctl。但是通过 `/etc/sysctl.conf` 来设置你喜欢的数值仍然是最好的，因为这样这些数值在重启以后仍然保留。

文件 `/etc/sysctl.conf` 只在OS X 10.3.9及以后的版本中遵守。如果你正在运行一个10.3.x之前的版本， 你必须编辑文件 `/etc/rc` 并在下列的命令中改变数值。

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

注意 `/etc/rc` 通常通过OS X系统更新重写，所以你应该预料到在每次更新后必须重做这些编辑。

在OS X 10.2以及更早的版本里，在 `/System/Library/StartupItems/SystemTuning/SystemTuning` 里编辑这些命令。

SCO OpenServer

缺省配置时，只允许每段 512KB 共享内存。要增大设置，首先进入 `/etc/conf/cf.d` 目录。要显示当前以字节记的 `SHMMAX` ， 运行：

```
./configure -y SHMMAX
```

设置 `SHMMAX` 的新值：

```
./configure SHMMAX=_value_
```

这里 `_value_` 是你想设置的以字节记的新值。设置完 `SHMMAX` 以后重新编译内核：

```
./link_unix
```

然后重启。

Solaris 2.6 到 2.9 (Solaris 6 到 Solaris 9)

相关的设置可以在 `/etc/system` 里面修改，例如：

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

你需要重启以使这些修改生效。又见http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html 获取关于老版本的Solaris共享内存的信息。

Solaris 2.10 (Solaris 10) 及以后 OpenSolaris

Solaris 10及以后的版本，以及OpenSolaris，缺省的共享内存和信号灯的设置对大多数PostgreSQL 应用来说是足够的。Solaris现在对 `SHMMAX` 的缺省是系统RAM的四分之一。 要进一步调整这些设置，使用一个和 `postgres` 用户相关的项目设置。例如，作为 `root` 运行下列命令：

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U post
```

这些命令增加 `user.postgres` 项目并且设置 `postgres` 用户的最大共享内存为8GB， 在下次用户登录时生效，或当你重启PostgreSQL（不是重新加载）生效。 上面的是假设PostgreSQL由在 `postgres` 组里面的 `postgres` 用户运行。 不需要服务器重启。

其他推荐的数据库服务器的内核设置修改将有大量的连接：

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

此外，如果你在一个区域里面运行PostgreSQL，你可能也需要提高区域资源使用的限制。 参阅*System Administrator's Guide*里面的"Chapter2: Projects and Tasks" 获取更多关于 `projects` 和 `prctl` 的信息。

UnixWare

在UnixWare 7 上，缺省配置里的最大共享内存段是 512kB 。 要显示 `SHMMAX` 的当前值，运行：

```
/etc/conf/bin/idtune -g SHMMAX
```

就会显示以字节记的当前的缺省的最小和最大值。要给 `SHMMAX` 设置一个新值，运行：

```
/etc/conf/bin/idtune SHMMAX _value_
```

`_value_` 是你想设置的以字节记的新值。设置完 `SHMMAX` 后，重建内核：

```
/etc/conf/bin/idbuild -B
```

然后重启。

17.4.2. 资源限制

Unix 类系统强制了许多资源限制，这些限制可能干扰PostgreSQL服务器的运行。 这里尤其重要是对每个用户的进程数目的限制、每个进程打开文件数目、以及每个进程可用的内存。 这些限制中每个都有一个"硬"限制和一个"软"限制。实际使用的是软限制， 但用户可以自己修改成最大为硬限制的数目。而硬限制是只能由 root 用户修改的限制。 系统调用 `setrlimit` 负责设置这些参数。shell 的内建命令 `ulimit` (Bourne shells) 或 `limit` (csh) 就是用于在命令行上控制资源限制的。 在 BSD 衍生的系统上， `/etc/login.conf` 文件控制在登录时对各种资源设置什么样的限制数值。 参阅操作系统文档获取细节。相关的参数是 `maxproc`，`openfiles`，`datasize` 。比如：

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

`-cur` 是软限制，后面附加 `-max` 就可以设置硬限制。

内核通常也有一些系统范围的资源限制。

- 在Linux上， `/proc/sys/fs/file-max` 决定内核可以支持的最大文件数。你可以通过往该文件写入一个不同的数值修改此值， 或者在 `/etc/sysctl.conf` 里增加一个赋值。 每个进程的最大打开文件限制是在编译内核的时候固定的； 参阅 `/usr/src/linux/Documentation/proc.txt` 获取更多信息。

PostgreSQL服务器每个连接都使用一个进程， 所以你应该至少允许和连接数相同的进程数，再加上系统其它部分所需要的数目。 通常这个并不是什么问题，但如果你在一台机器上运行多个服务器，那你就要把事情理清楚。

打开文件数目的出厂缺省设置通常设置为"社会友好"数值， 就是说允许许多用户共存一台机器，而不会导致系统资源使用的不当比例。 如果你在一台机器上运行许多服务器，这也许就是你想要的，但是在特殊的服务器上， 你可能需要提高这个限制。

问题的另外一边，一些系统允许独立的进程打开非常多的文件；如果有几个进程这么干， 那系统范围的上限就很容易达到。如果你发现这样的现像，并且不想修改系统范围的限制， 你就可以设置PostgreSQL的max_files_per_process 配置参数来限制打开文件数的消耗。

17.4.3. Linux 内存过提交

在Linux 2.4 以及之后的版本里，缺省的虚拟内存的行为不是对PostgreSQL最优的。 原因在于内核实现内存过提交的方法，如果其它进程的内存请求导致系统用光虚拟内存， 那么内核可能会终止PostgreSQL主服务器进程。

如果发生了这样的事情，你会看到像下面这样的内核信息(参考你的系统文档和配置， 看看在哪里能看到这样的信息)：

```
Out of Memory: Killed process 12345 (postgres).
```

这表明 `postgres` 因为内存压力而终止了。尽管现有的数据连接将继续正常运转， 但是新的连接将无法接受。要想恢复，你应该重启PostgreSQL。

一个避免这个问题的方法是在一台你确信不会因为其它进程而耗尽内存的机器上运行PostgreSQL。

如果PostgreSQL本身是系统耗尽内存的原因，你可以通过改变你的配置来避免这个问题。 在某些情况下，这样可能帮助降低内存相关的配置参数，尤其是 `shared_buffers` 和 `work_mem` 。其他情况下， 这个问题可能是由于允许太多的连接到数据库服务器本身引起的。在许多情况下， 减少 `max_connections` 而不是利用外部连接池的软件会更好。

在 Linux 2.6 以及以后的版本里，可以修改内存的行为，这样它就不会再"过提交"内存。 尽管这个设置将不会完全阻止OOM killer 被引用，然是它将显著地减少并且将因此导致更稳健的系统行为。 这是通过用 `sysctl` 选取一个严格的过提交模式实现的：

```
sysctl -w vm.overcommit_memory=2
```

或者在 `/etc/sysctl.conf` 里放一个等效的条目。你可能还希望修改相关的 `vm.overcommit_ratio` 设置。详细信息请参阅内核文档的 `Documentation/vm/overcommit-accounting` 文件。

另外一种方法，改变或不改变 `vm.overcommit_memory` 都可以使用， 设置与进程相关的 `oom_score_adj` 值为主进程 `-1000` ， 从而保证它不会成为OOM killer的目标。 最简单的方法是在调用postmaster之前在postmaster的启动脚本执行：

```
echo -1000 > /proc/self/oom_score_adj
```

请注意，这个操作必须由root来做，否则将不会有任何作用；所以一个root所有的启动脚本是做这个最简单的地方。 如果你这样做，你可能也希望用 `-DLINUX_OOM_SCORE_ADJ=0` 添加到 `CPPFLAGS` 来建立PostgreSQL。这将导致主子进程以标准 `oom_score_adj` 值0来运行， 所以OOM killer仍然可以在需要时以它们为目标。

老版本的Linux内核不提供 `/proc/self/oom_score_adj` ，但是可能有相同功能的名为 `/proc/self/oom_adj` 的以前的版本。除了禁用值为 `-17` 而不是 `-1000` 外，它们做相同的工作。相应的PostgreSQL的建立标识为 `-DLINUX_OOM_ADJ=0` 。

> **Note:** 有些供应商的 Linux 2.4 内核有着早期 2.6 过提交的 `sysctl` 。不过， 在没有相关代码的2.4内核里设置 `vm.overcommit_memory` 为 2 只会让事情更糟， 而不是更好。我们建议你检查一下实际的内核源代码(参阅文件 `mm/mmap.c` 里面的 `vm_enough_memory` 函数)，核实一下这个是在你的内核里存在的， 然后再在 2.4 内核里使用这个特性。文档文件 `overcommit-accounting` 的存在不能当作是这个特性存在的证明。如果有问题，请询问你的内核供应商的专家。

17.5. 关闭服务器

有好几种关闭数据库服务器的方法。通过给 `postgres` 进程发送不同的信号， 你就可以控制关闭服务器的不同方法。

SIGTERM

这是智能关闭模式。接收到SIGTERM以后， 服务器不再允许新的连接，但是允许所有活跃的会话正常完成他们的工作， 只有在所有会话都结束任务后才关闭。如果服务器处在在线备份模式，它另外等待在线备份模式不再活跃。 当备份模式活跃时，将仍允许新的连接，但是只针对超级用户（这个例外允许超级用户连接以终止在线备份模式）。 如果当请求智能关闭时服务器正在恢复，那么恢复和流复制将在所有普通会话终止后停止。

SIGINT

这是快速关闭模式。不再允许新的连接，向所有活跃服务器发送SIGTERM (让它们立刻退出)，然后等待所有子进程退出并关闭数据库。如果服务器处在在线备份模式， 备份模式将终止，使得备份无效。

SIGQUIT

这是立即关闭模式。主 `postgres` 进程将向所有子进程发送 SIGQUIT并且立即退出(所有子进程也会立即退出)，而不会妥善地关闭数据库系统。 这样做会导致下次启动时的恢复(通过重放 WAL 日志)。我们推荐只在紧急的时候使用这个方法。

pg_ctl程序提供了一个发送这些信号关闭服务器的便利接口。 另外，你在非Windows系统上可以用 `kill` 直接发送这些信号。 可以用 `ps` 命令或者从数据目录里的 `postmaster.pid` 文件中找出 `postgres` 的PID。所以，举例来说，要做一次快速关闭：

```
$ <kbd class="literal">kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`</kbd>
```

> **Important:** 最好不要用SIGKILL来关闭服务器。这样做将阻止服务器释放共享内存和信号灯， 那样的话你只能在新服务器启动前自己手动做这件事。另外，SIGKILL直接杀死 `postgres` 进程而不等它传递信号给子进程，所以我们也将必须手动的杀死独立的子进程。

当允许其他会话继续时终止一个独立的会话，使用 `pg_terminate_backend()` (参见Table 9-59)或发送一个SIGTERM 信号到与这个会话有关的子进程。

17.6. 升级一个 PostgreSQL 集群

本节讨论怎样从一个PostgreSQL发布到一个新的发布升级你的数据库数据。

PostgreSQL主要版本由前两个数字的版本号代表，例如8.4。 PostgreSQL主要版本由前三个数字的版本号代表，例如8.4.2是第二个8.4的维护性发布。 维护性发布从不改变内部存储器格式并且总是与之前的和之后的相同维护性版本号的维护性版本兼容， 例如8.4.2兼容8.4,8.4.1和8.4.6。要在兼容的版本之间更新，在服务器关闭时简单的替换可执行文件并且重启服务器就可以。 数据目录保留不改变—维护性升级就这么简单。

对PostgreSQL的主要发布，内部数据存储格式是改变的主题， 因此并发升级。移动数据到一个新的主要版本的传统的方法是转储并重新加载数据库。 就像下面描述的那样，其他方法也适用。

新的主要版本还通常介绍一些用户可见的不兼容性，所以可能需要改变应用程序。 所有用户可见的改变都在版本注释里面列出（Appendix E）； 尤其注意章节标签"Migration"。如果你正在几个主要版本间升级， 一定要阅读每个中间版本的版本注释。

谨慎的用户可能想在全面切换前在新的版本上测试他们的客户端应用；因此， 设置并发新旧版本的安装是一个好的主意。当测试一个PostgreSQL的主要升级时， 考虑下列类别的可能变化：

Administration

在每个主要版本中，管理员对服务器可用的监视和控制功能通常会改变和增加。

SQL

通常包含新的SQL命令功能而不是行为上的改变，除非在版本注释里面特别提到。

Library API

像libpq这样典型的库只添加新的函数，同样除非在版本注释里面特别提到。

System Catalogs

系统目录变更通常只影响数据库管理工具。

Server C-language API

这涉及到后端函数API的变化，它是用C编程语言写的。这样的改变影响引用深层服务器后端函数的编码。

17.6.1. 通过pg_dump升级数据

要从PostgreSQL的一个主要版本转储数据并加载到另一个版本，必须使用pg_dump; 文件系统级备份方法将不能使用。（有检查防止你用不兼容的PostgreSQL 版本使用数据目录，所以在一个数据目录上尝试启动错误的服务器版本也不会造成大的伤害。）

建议使用新版本的PostgreSQL pg_dump和pg_dumpall程序， 利用这些程序可能的增强功能。当前转储程序版本可以从一直到7.0的任何服务器版本中读取数据。

这些说明假设现有的安装在 `/usr/local/pgsql` 目录下，数据区域在 `/usr/local/pgsql/data` 目录下。适当的替代你的路径。

1. 如果做一个备份，确保数据库没有被更新。这不影响备份的完整性，但是将不会包括改变了的数据。 如果必须要这样做，那么编辑文件 `/usr/local/pgsql/data/pg_hba.conf` (或相等的)里的权限， 设置除了你之外的人都不可以访问。参阅Chapter 19获取关于访问控制的更多信息。

   备份你的数据库安装，输入：

   ```
   &lt;kbd class="literal"&gt;pg_dumpall &gt; `_outputfile_`&lt;/kbd&gt;
   ```

   如果您需要保存OID（例如当使用它们作为外键时），那么在运行pg_dumpall 时使用 `-o` 选项。

   要做备份，可以使用当前运行版本的pg_dumpall命令。然而， 要获取最佳结果尝试使用PostgreSQL 9.3.1的 pg_dumpall命令，因为这个版本包含bug修复和对老版本的改进。 虽然在你还没有安装新版本时这个建议看起来很特别，但是如果你打算同时安装新版本和老版本， 那么这个建议是可行的。在这种情况下你可以正常完成安装然后传送数据。 这也将减少停机时间。

2. 关闭旧的服务器：

   ```
   &lt;kbd class="literal"&gt;pg_ctl stop&lt;/kbd&gt;
   ```

   系统上的PostgreSQL在开机时启动，可能有一个启动文件将完成同样的事情。 例如，在Red Hat Linux系统上，可能发现这个在工作：

   ```
   &lt;kbd class="literal"&gt;/etc/rc.d/init.d/postgresql stop&lt;/kbd&gt;
   ```

   参阅Chapter 17获取关于启动和停止服务器的详细信息。

3. 如果从备份中恢复，重命名或者删除旧的安装目录。重命名目录是个好主意，而不是删掉它， 以防你有困难并且需要恢复它。记住，目录可能会占用大量磁盘空间。要重命名目录，使用像下面这样的命令：

   ```
   &lt;kbd class="literal"&gt;mv /usr/local/pgsql /usr/local/pgsql.old&lt;/kbd&gt;
   ```

（一定要把目录作为一个单元，所以相对路径保持不变。）

4. 像Section 15.4. 中概述的那样安装PostgreSQL新版本。
5. 如果需要创建一个新的数据库集群。请记住，当登陆到特定的数据库用户账户 （如果已经升级，那么已经有这个账户了）时必须执行下面的命令。

```
&lt;kbd class="literal"&gt;/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data&lt;/k
```

6. 恢复以前的 `pg_hba.conf` 和任意 `postgresql.conf` 修改。
7. 启动数据库服务器，再次使用特定的数据库用户账户：

```
&lt;kbd class="literal"&gt;/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data&lt;
```

8. 最终，从备份中恢复数据：

```
&lt;kbd class="literal"&gt;/usr/local/pgsql/bin/psql -d postgres -f `_outputfile_`&lt
```

   使用新的 psql

最少的停机时间可以通过在一个不同的目录下安装新的服务器，并且并行在不同的端口运行旧的和新的服务器来达到。 然后就可以使用类似：

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

来传送数据。

17.6.2. Non-Dump 升级方法

pg_upgrade模块允许安装从一个主PostgreSQL版本到下一个的迁移。 升级可以在几分钟内完成。

还可以使用特定的复制方法，比如Slony，创建一个更新的PostgreSQL 版本备用服务器。这是可能的，因为Slony支持不同PostgreSQL主版本间的复制。 备机可以在相同的计算机上，也可以在不同的计算机上。一旦它与主服务器（在老PostgreSQL 版本上运行）同步，就可以切换主机使备机成为主机，并立即关闭旧的数据库。这样的切换使升级只有几秒钟的关机时间。

17.7. 防止服务器欺骗

当服务器正在运行时，恶意用户代替正常的数据库服务器是不可能的。然而，当服务器关闭时， 本地用户通过启动他们自己的服务器欺骗正常的服务器是可能的。行骗服务器可以读取客户端发来的查询和密码， 但不能返回任何数据，因为 `PGDATA` 目录因为目录存取权限仍然是安全的。 因为任何用户都可以启动一个数据库服务器，所以欺骗是可能的；客户端不能识别无效的服务器， 除非它是专门配置的。

为 `local` 连接防止欺骗的最简单的方法是使用Unix域套接字目录 (unix_socket_directories)，这个目录只为受信任的本地用户写入了权限。 如果你担心一些应用可能仍然参考 `/tmp` 为套接字文件，并且因此很容易受欺骗， 那么在操作系统启动时创建一个符号链接 `/tmp/.s.PGSQL.5432` 指向重定位的套接字文件。 您还可能需要修改 `/tmp` 清理脚本以防止删除符号链接。

要在TCP连接上防止欺骗，最好的解决方法是使用SSL认证，并确保客户检查服务器的证书。 要做到这一点，服务器必须配置为只接受 `hostssl` 连接(Section 19.1)， 并且有SSL秘钥和证书文件(Section 17.9)。TCP客户必须使用 `sslmode=verify-ca` 或 `verify-full` 连接，并且有适当的根证书文件安装(Section 31.18.1)。

17.8. 加密选项

PostgreSQL提供了几个不同级别的加密， 并且在保护数据库服务器不受数据窃贼、不道德管理员、不安全网络等因素泄漏的方面提供很高的灵活性。 加密可能也是保护一些诸如医疗记录和财务交易等敏感数据的要求。

口令存储加密

缺省的时候，数据库用户的口令以 MD5 散列的方式存储，所以管理员无法判断赋予用户的实际口令。 如果 MD5 加密用于客户端认证，那么未加密的口令甚至都不可能临时出现在服务器上， 因为客户端在透过网络发送之前，就先用 MD5 加密了。

为指定的字段加密

pgcrypto模块允许对某些字段进行加密存储。这个功能在某些数据是敏感的情况下有用。 客户端提供解密的密钥，然后数据在服务器端解密，然后发送给客户端。

在数据解密和数据在服务器与客户端之间传递时，解密数据和解密密钥将会在服务器端存在短暂的一段时间。 这就给那些可以完全访问数据库服务器的人提供了一个短暂的截获密钥和数据的时间， 这样的人一般是数据库管理员。

数据库分区加密

在 Linux 上，加密可以在使用"回环设备"(loopback device)挂载的文件系统上面进行。 这样就可以把磁盘上整个文件分区都加密，然后由操作系统解密。在 FreeBSD 上， 等效的设施叫 GEOM 基本磁盘加密(gbde)，并且许多其他操作系统支持这个功能，包括Windows。

这个机制避免了在整个计算机或者磁盘驱动器被窃的情况下，未加密的数据被从驱动器中读取。 它无法防止在文件系统被挂载的时候的攻击，因为在挂载之后，操作系统提供数据的解密视图。 不过，要想挂载文件系统，你需要有一些方法把解密密钥传递给操作系统， 有时候这个密钥就存储在挂载该磁盘的主机的某个地方。

跨网络加密口令

`MD5` 认证方法在客户端将口令发给服务器之前双重加密之。第一次 MD5 加密是基于用户名的， 然后在连接数据库的时候，用服务器发送的随机盐粒再次加密。 这个双重加密的数值就是从网络传递给服务器的数值。双重加密不仅可以避免口令泄漏， 还可以避免稍后其它的连接使用同样的加密口令连接数据库(回放攻击)。

透过网络加密数据

SSL 连接加密所有透过网络发送的数据：口令、查询、返回的数据。 `pg_hba.conf` 文件允许管理员声明哪些主机可以使用不加密的连接(`host`)， 以及哪些主机需要使用 SSL 加密的连接(`hostssl`)。客户也可以指定只通过SSL连接到服务器。 我们也可以使用Stunnel或SSH加

密数据传输。

SSL 主机认证

客户端和主机都可以提供 SSL 证书给对方。这么做需要在两边都进行一些额外的配置工作，但是这种方式提供了比简单使用用户名和口令更强的身份认证的手段。 它避免一个计算机装作是服务器，然后读取客户端口令， 只要时间长得足够读取客户端发送的口令就行了。它还避免了"中间人" 攻击(在客户端和服务器之间有台计算机，伪装成为服务器并且读取然后将所有数据在客户端和服务器之间传递)。

客户端加密

如果服务器机器的系统管理员是不可信的，那么客户端加密数据也是必要的；这种情况下，未加密的数据从来不会在数据库服务器上出现。数据在发送给服务器之前加密， 而数据库结果必须在客户端使用之前解密。

17.9. 用 SSL 进行安全的 TCP/IP 连接

PostgreSQL有一个内建的通过SSL进行加密的客户端/服务器端的通讯， 这样可以增加安全性。这个特性要求在客户端和服务器端都安装OpenSSL 并且在编译PostgreSQL的时候打开(参阅Chapter 15)。

当编译了SSL进去以后，可以通过将 `postgresql.conf` 中的ssl 设置为 `on` 打开PostgreSQL服务器的SSL支持。 服务器将在同一个 TCP 端口上同时监听标准的和 SSL 的连接， 并且将与任何正在连接的客户端进行协商，协商是否使用SSL 。缺省时， 这是根据客户端的选项而定的。参阅Section 19.1 获取如何强制服务器端只使用SSL进行某些或者全部连接的信息。

PostgreSQL读取系统范围的OpenSSL 配置文件。缺省的，这个文件名为 `openssl.cnf` ，位于 `openssl version -d` 报告的目录中。这个缺省可以通过设置环境变量 `OPENSSL_CONF` 为所需配置文件的名称重写。

OpenSSL支持多种不同强度的密码和认证算法。 当一列密码可以在OpenSSL配置文件中指定时， 可以通过修改 `postgresql.conf` 中的ssl_ciphers， 指定数据库服务器专用的密码。

> **Note:** 使用 `NULL-SHA` 或 `NULL-MD5` 可能有身份验证开销而没有加密开销。 不过，中间人能够读取和通过客户端和服务器的通信。此外，加密开销相比身份认证的开销是极小的。 有这些原因建议不使用NULL加密。

要在SSL模式中启动服务器，必须包含包含服务器证书和私钥的文件。缺省的， 这些文件分别命名为 `server.crt` 和 `server.key` ，存在于服务器的数据目录里， 但是其他名字和位置可以用配置参数ssl_cert_file和ssl_key_file 指定。在Unix系统， `server.key` 的权限禁止任何world或group访问； 通过命令 `chmod 0600 server.key` 来做。如果私钥受密码保护，服务器将会提示输入密码， 将会等到输入后启动。

在有些情况下，服务器证书可能由一个"中间"认证授权签名，而不是直接由受信任的客户端。若要使用这样的证书，请追加授权签名证书到 `server.crt` 文件，然后其父颁发机构的证书，然后直到一个客户端信任的"根"授权。 `server.crt` 包含多个证书， 在任何情况下都应包括根证书。

17.9.1. 使用客户端证书

如果需要客户端提供受信任的证书，把认证中心(CAs)的证书放在你信任的在数据目录下的 `root.crt` 文件里，并且设置 `postgresql.conf` 里的参数 ssl_ca_file为 `root.crt` ，设置 `pg_hba.conf` 里适当的 `hostssl` 行参数 `clientcert` 为1。 然后将在SSL连接启动时从客户端请求该证书。 （Section 31.18描述了如何在客户端安装证书。）服务器将验证客户端的证书

是由受信任的认证中心之一签发的。如果参数ssl_crl_file 也设置了，那么证书撤销列表(CRL)项也要检查。 (参阅http://h71000.www7.hp.com/DOC/83final/BA554_90007/ch04s02.html 图标显示SSL证书的使用。)

在 pg_hba.conf 文件中 clientcert 选项对于所有的认证方法都可用， 但仅适用 hostssl 指定的行。当 clientcert 没有指定或设置为0， 服务器将仍然对其CA列表验证提供的客户端证书，如果配置了，它将不会坚持提交客户端证书。

请注意 root.crt 列出顶级的CA，这些CA被认为是受签约客户端证书信任的。 原则上不需要列出标记服务器证书的CA，尽管在大多数情况下，客户端证书也将信任的该CA。

如果你设置客户端证书，你可能希望用 cert 认证方法，因此使证书控制用户身份验证， 以及提供连接安全。参阅Section 19.3.10获取详细信息。

17.9.2. SSL服务器文件的使用

Table 17-2概述了与在服务器上设置SSL相关的文件。 （显示的文件名字是默认的或典型的名字。 本地配置的名字可能会不同。）

Table 17-2. SSL服务器文件的使用

文件	内容	作用
ssl_cert_file ($PGDATA/server.crt)	服务器证书	发送到客户端标识服务器的身份
ssl_key_file ($PGDATA/server.key)	服务器私钥	证明服务器证书是由所有者发送，并不表示证书所有者是可信的
ssl_ca_file ($PGDATA/root.crt)	受信任的认证中心	检查该客户端证书由受信任的认证中心签署
ssl_crl_file ($PGDATA/root.crl)	由认证中心吊销的证书	客户端证书不能在此列表中

文件 server.key ， server.crt ， root.crt 和 root.crl （或他们配置的供选择的名字）仅在服务器启动时检查； 如果你修改了它们，那么必须重启服务器才能生效。

17.9.3. 创建自签名的证书

要为服务器创建一个快速自认证的证书，使用下面的OpenSSL命令：

```
openssl req -new -text -out server.req
```

填充那些openssl向你询问的信息。确保把本地主机名当做"Common Name"输入。 要求的密码可以留空。该程序将生成一把用口令保护的密钥。小于四字符的口令保护是不被接受的。要移去密钥(如果你想自动启动服务器就得这样)，运行下面的命令：

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

输入旧口令把现有密钥解锁。然后:

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

将证书变成自签名的证书，并且复制密钥和证书到服务器能找到的地方。最后执行操作：

```
chmod og-rwx server.key
```

因为如果其权限比这更高，服务器将拒绝该文件。更多关于怎样创建服务器私钥和证书的细节， 请参考OpenSSL文档。

你可以用一个自认证的证书进行测试，但是在生产环境中应该使用一个由认证中心(CA， 全球的CA或者区域的CA都可以)签发的证书， 这样客户端才能够识别服务器的身份。如果所有客户都是本地组织的，建议使用本地CA。

17.10. 用SSH隧道进行安全 TCP/IP 连接

使用SSH对PostgreSQL 服务器和客户端之间的网络连接进行加密是可能的。经过适当处理后， 这样做可以获得一个足够安全的网络连接。即使是没有 SSL 的客户端上也如此。

首先确认SSH正在和PostgreSQL 服务器的同一台机器上正确地运行，而且你可以通过某个用户用 `ssh` 登录。 然后你可以用下面这样的命令从客户端的机器上建立一个安全通道：

```
ssh -L 63333:localhost:5432 joe@foo.com
```

`-L` 参数的第一个数字(63333)是你这端通道的端口号，可以是任何未使用的端口。 （IANA提供端口49152到65535位私人使用。）第二个数字(5432)是通道的远端， 也就是服务器使用的端口号。在两个端口号之间的名称或者 IP 地址是你准备连接的数据库服务器， 在例子中是 `foo.com` 。为了使用这个通道与数据库服务器连接， 你在本机于端口63333连接：

```
psql -h localhost -p 63333 postgres
```

对于数据库服务器而言，在这种情况下，它会把你当做主机 `foo.com` 连接到 `localhost` 的真正的用户 `joe` ， 并且使用为这个用户和主机设置的认证手段进行认证。请注意，服务器不会认为连接是 SSL 加密的， 因为实际上在SSH服务器和PostgreSQL 服务器之间是没有加密的。只要它们在同一台机器上，这么做并不会导致任何安全漏洞。

为了保证能够成功地建立通道，你必须被允许作为 `joe@foo.com` 通过 `ssh` 建立连接，就像你使用 `ssh` 创建终端会话一样。

你也可以设置端口转发

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

但是随后数据库服务器将看到连接从 `foo.com` 界面进来， 这是不被缺省设置 `listen_addresses = 'localhost'` 开放的。 这通常不是你想要的。

如果你必须通过某些登陆主机"跳跃"到数据库服务器， 一个可能的设置看起来像这样：

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

请注意，这个方式从 `shell.foo.com` 到 `db.foo.com` 的连接将不会通过SSH通道加密。当网络以各种方式被限制时，SSH提供了相当多的配置可能性。 请参阅SSH文档获取详情。

> **Tip:** 还有几种不同的产品可以提供安全的通道，所使用的过程类似我们刚刚描述的过程。

17.11. 在Windows上注册事件日志

要注册操作系统的Windows事件日志库，发出这个命令：

```
<kbd class="literal">regsvr32 `_pgsql_library_directory_`/pgevent.dll</kbd>
```

这将使用事件查看器创建注册表条目，在默认的名为 `PostgreSQL` 的事件源下。

要指定一个不同的事件源名字（参阅event_source），使用 `/n` 和 `/i` 选项：

```
<kbd class="literal">regsvr32 /n /i:`_event_source_name_` `_pgsql_library_directory_`/pge
```

要从操作系统中注销event log库，发出这个命令：

```
<kbd class="literal">regsvr32 /u [/i:`_event_source_name_`] `_pgsql_library_directory_`/p
```

> **Note:** 要在数据库服务器中启用事件日志记录，在 `postgresql.conf` 中修改log_destination包含 `eventlog` 。

Chapter 18. 服务器配置

Table of Contents

有一堆配置参数可以影响数据库系统的行为。本章第一节我们将描述一下如何设置它们。 然后随后的小节我们将逐个讨论它们。

18.1. 设置参数

18.1.1. 参数名和值

所有参数名都是大小写不敏感的。每个参数都可以接受四种类型之一： 布尔、整数、浮点数、字符串。布尔值可以是(都是大小写无关) `on` , `off` , `true` , `false` , `yes` , `no` , `1` , `0` 或这些东西的任意清晰无歧义的前缀。

一些设置指定内存或时间值，其隐含的单位可能是： kB(千字节)、块(通常是8KB)、毫秒、秒、分钟等等。 隐含单位可以通过引用 `pg_settings` . `unit` 获取。为了避免混淆， 可以在指定数值的同时指定单位。可用内存单位： `kB` (千字节), `MB` (兆字节), `GB` (吉字节)； 可用时间单位： `ms` (毫秒), `s` (秒), `min` (分钟), `h` (小时), `d` (天)。 内存单位中的"千"等于1024， 而不是1000。

作为字符串参数的同样方式指定"枚举"类型参数，但被限制在值的有限集合中。 可以从 `pg_settings` . `enumvals` 找到允许的值。 枚举参数值是不区分大小写的。

18.1.2. 通过配置文件设置参数

设置这些参数的一个方法是编辑 `postgresql.conf` 文件， 它通常在数据目录里(当数据库集群目录初始化的时候，会在那里安装一个缺省拷贝)。比如， 下面是一个该文件的例子：

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

参数是每条一行。选项名和值之间的等号是可选的。 空白和空行被忽略。井号(`#`)引入注释。 非简单标识符或者数字必须用单引号包围。 如果需要在参数值里嵌入单引号，要么写两个单引号(推荐方法)，要么用反斜扛包围。

主服务器进程每次收到SIGHUP信号 后都会重新读取这个配置文件， 最简单的发送方法就是使用来自命令行的 `pg_ctl reload` 或者调用SQL函数 `pg_reload_conf()` 。 同时主服务器进程也将这个信号广播给所有正在运行的服务器进程， 这样现有会话也能得到新值。另外，你可以只向一个服务器进程直接发送信号。 有些参数只能在服务器启动的时候设置；对这些条目的修改将被忽略， 直到下次服务器重启。配置文件中的无效参数设置在SIGHUP处理中也被忽略（但已登录）。

18.1.3. 设置参数其他的方法

第二种设置这些配置参数的方法是把它们作为命令行参数传递给 `postgres` ，比如：

```
postgres -c log_connections=yes -c log_destination='syslog'
```

命令行选项覆盖 `postgresql.conf` 中的矛盾设置。请注意， 这意味着你不能通过编辑 `postgresql.conf` 在运行时改变其数值，因此， 虽然命令行方法很方便，但会付出灵活性的代价。

有时候，给某一个特定会话一个命令行参数也是很有用的。 可以在客户端使用环境变量 `PGOPTIONS` 来实现这个目的：

```
env PGOPTIONS='-c geqo=off' psql
```

(可以用于任何基于libpq的客户端应用， 不光是psql)： 请注意，这个变量对那些需要在服务器启动后固定的选项或者 必须在 `postgresql.conf` 里声明的选项是无效的。

并且，我们可以给一个用户或者一个数据库赋予一套选项设置。 在一个会话开始的时候，装载所涉及到的用户和数据库的缺省设置。 命令ALTER ROLE和ALTER DATABASE 分别用于配置这些设置。 针对每个数据库的设置将覆盖任何从 `postgres` 命令行或者配置文件收到的设置， 然后接着又被针对每个用户的设置覆盖； 最后又会都被针对每个会话的设置覆盖。

一些选项可以在独立的SQL会话中修改，方法是使用SET命令，比如：

```
SET ENABLE_SEQSCAN TO OFF;
```

如果允许用 `SET` 设置，这种针对每个数据库的设置将覆盖任何来自其它方面的设置。 有些参数不能通过 `SET` 改变：比如，如果这些选项不重启动PostgreSQL就无法合理控制其行为。 同样，有些参数只能由超级用户通过 `SET` 或 `ALTER` 修改，而普通用户不能修改。

18.1.4. 检查参数设置

SHOW命令检查所有参数的当前值。

我们也可以用虚表 `pg_settings` 来显示和更新当前会话的运行时参数。当它们改变时，参见Section 47.66获取详细信息，以及不同变量类型的描述。 `pg_settings` 等效于 `SHOW` 和 `SET` ，但是用起来更方便，因为它可以和其它表连接起来使用， 或者用任意用户需要的选择条件来查询。 它还包含了来自 `SHOW` 的每个参数的更多详细信息。

18.1.5. 配置文件包含

除了参数设置外， `postgresql.conf` 文件还包含*include*指令， 它声明了另外一个文件的读取和处理，正如在这一点上插入到配置文件。 这个特性允许配置文件被分成物理上独立部分。 Include指令看起来像：

```
include 'filename'
```

如果文件名不是绝对路径，那么它被看成包含引用配置文件目录的相对路径。 可以进行嵌套。

此外，还有一个 `include_if_exists` 指令，它的作用和 `include` 指令是相同的，除了被引用的文件不存在或无法读取时的行为。 规则的 `include` 会认为这是一个错误条件，但 `include_if_exists` 只是记录一条消息，并继续处理引用的配置文件。

`postgresql.conf` 文件也包含 `include_dir` 指令，声明了配置文件的 整个目录，它的类似用法：

```
include_dir 'directory'
```

非绝对路径遵循和单个文件包括指令相同的规则， 它们是相对于包含引用的配置文件目录。 在该目录中，唯一一个非目录的文件， 其名以后缀 `.conf` 的将被包括在内。 文件名以 `.` 字符开头的被排除在外， 为防止出错，它们都隐藏在一些平台上。在include目录中的多个文件按照文件名的顺序进行处理。 文件名按照C语言环境的规则排序。字母前的数字和小写字母前大写字母。

包含文件或目录可用于数据库配置逻辑上独立的部分， 而不是单一的 `postgresql.conf` 文件。 考虑有两个数据库服务器公司， 每一个有不同的内存量。 有可能有配置都共享的元素，比如日志。 但是服务器内存相关参数两者之间不同。 也有可能是服务器特定的自定义。 管理这种情况的方法是打破了自定义配置更改为你的网站的三个文件。 你可以添加这些到你的 `postgresql.conf` 文件末尾，包括：

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

所有系统可能有同样的 `shared.conf` 文件。具有特定内存量的每个服务器可以共享 相同的 `memory.conf` ；可能有8GB内存的服务器， 另一个是16GB。 最后 `server.conf` 可能真正有服务器特定配置文件信息。

另外一个可能是创建配置文件目录，并且将这些信息放入文件中。 比如， `conf.d` 目录可能在 `postgresql.conf` 末尾被引用：

```
include_dir 'conf.d'
```

那么你可以像这样在`conf.d`目录中命名文件：

```
00shared.conf
01memory.conf
02server.conf
```

这显示了文件被加载的明确顺序。这是非常重要的，因为当服务器正在读取它使用的配置时，只读取最后的设置。这个例子中的`conf.d/02server.conf`的一些设置会覆盖`conf.d/01memory.conf`中设置的值。

你可能会使用这种配置目录的方法，当命名这些文件更加详细时：

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

这样的安排使每个配置文件的变化具有唯一名称。当一些服务器的配置都存储在一个地方的时候，这可以帮助消除歧义。比如版本 控制存储（版本控制下存储数据库配置文件是另一种很好的做法）。

18.2. 文件位置

除了已经提到过的 `postgresql.conf` 文件之外，PostgreSQL还使用另外两个手工编辑的配置文件， 它们控制客户端认证(在Chapter 19里讨论)。缺省时，所有三个配置文件都存放在数据库集群的数据目录里。 本节描述的选项允许配置文件放在别的地方。这么做可以简化管理，特别是如果配置文件独立放置，通常可以很容易保证它得到恰当的备份。

`data_directory` (`string`)

声明为数据存储使用的目录。这个选项只能在服务器启动的时候设置。

`config_file` (`string`)

声明主服务器配置文件(通常叫 `postgresql.conf`)。这个选项只能在 `postgres` 命令行上设置。

`hba_file` (`string`)

声明基于主机的认证(HBA)配置文件(通常叫 `pg_hba.conf`)。这个选项只能在服务器启动的时候设置。

`ident_file` (`string`)

声明用于Section 19.2用户名匹配的配置文件(通常叫 `pg_ident.conf`)。 这个选项只能在服务器启动的时候设置。

`external_pid_file` (`string`)

声明可被服务器管理程序使用的额外PID文件。这个选项只能在服务器启动的时候设置。

缺省安装不会明确设置这些参数。而是用命令行参数 `-D` 或者环境变量 `PGDATA` 声明数据目录， 并将上述配置文件都放在数据目录里。

如果你想把配置文件放在别的地方，那么 `postgres` 的命令行参数 `-D` 或者环境变量 `PGDATA` 必须指向包含配置文件的目录，而 `postgresql.conf` 里(或者命令行上)的 `data_directory` 选项 必须设置为数据目录实际存放的地方。请注意， `data_directory` 将覆盖 `-D` 和 `PGDATA` 指定的数据目录， 但是不覆盖配置文件的目录。

如果你愿意，可以使用选项 `config_file` ， `hba_file` 和/或者 `ident_file` 分别声明配置文件的路径。 `config_file` 只能在 `postgres` 命令行上设置， 但是其它的可以在主配置文件里设置。 如果明确设置了所有三个选项和 `data_directory` ，那么就没必要声明 `-D` 或者 `PGDATA` 。

在设置任何这些选项的时候，相对路径将被解释为相对于 `postgres` 启动时候的路径。

18.3. 连接和认证

18.3.1. 连接设置

`listen_addresses` (`string`)

声明服务器监听客户端连接的TCP/IP地址。值是一个逗号分隔的主机名和/或数字IP地址。 特殊项 `*` 对应所有可用IP接口。 `0.0.0.0` 允许监听所有IPV4地址，并且 `::` 允许监听所有IPV6地址。 如果这个列表是空的，那么服务器不会监听任何IP接口， 这种情况下，只有Unix域套接字可以用于连接数据库。 缺省值localhost只允许进行本地"回环"连接。 客户端认证(Chapter 19)允许细粒度控制谁能访问服务器， `listen_addresses` 控制尝试连接哪个接口。 这个参数只能在服务器启动的时候设置。

`port` (`integer`)

服务器监听的TCP端口；缺省是5432。请注意同一个端口号用于服务器监听的所有IP地址。 这个参数只能在服务器启动的时候设置。

`max_connections` (`integer`)

允许和数据库连接的最大并发连接数。缺省通常是100， 但是如果内核设置不支持这么大(在initdb的时候判断)，可能会比这个数少。 这个参数只能在服务器启动的时候设置。

当运行一个备用服务器，你必须将此参数设置为比主服务器上相同的或更高的值。否则， 不允许在备用服务器上进行查询。

`superuser_reserved_connections` (`integer`)

决定为PostgreSQL超级用户连接而保留的连接"槽位"。 一次最多可以同时激活max_connections个连接。在活跃的并发连接数接到了 `max_connections` 减去 `superuser_reserved_connections` 的时候，新的连接就只能由超级用户发起了， 并且不接收新的复制连接。

缺省值是3。这个值必须小于 `max_connections` 的值。 这个参数只能在服务器启动的时候设置。

`unix_socket_directories` (`string`)

指定Unix域套接字（S）的目录， 并且服务器监听来自客户端应用程序的连接。 多个套接字可以通过列出多个用逗号分隔的目录创建。 忽略条目之间的空白; 如果你需要包含空格或逗号的名字，增加带有双引号的目录名。 空值指定不监听任何Unix域的套接字， 在这种情况下，只有TCP/IP套接字可以用于连接到服务器。 默认值通常是 `/tmp` ， 但可以在编译的时候改变。 这个参数只能在服务器启动的。

除套接字文件本身外，它被命名为 `.s.PGSQL.``_nnnn_` ，其中 `_nnnn_` 是服务器的端口号，普通文件 命名的 `.s.PGSQL.``_nnnn_` .lock 将在每个 `unix_socket_directories` 目录中创建。 永远都不应该手动删除这两个文件。

这个参数在Windows上是无关紧要的，它不具有Unix域套接字。

`unix_socket_group` (`string`)

设置Unix域套接字的所属组(套接字的所属用户总是启动服务器的用户)。 可以与选项 `unix_socket_permissions` 一起用于对套接字进行访问控制。 缺省是一个空字符串， 表示当前服务器用户的缺省组。这个选项只能在服务器启动的时候设置。

这个参数在Windows上是无关紧要的，它不具有Unix域套接字。

`unix_socket_permissions` (`integer`)

设置Unix域套接字的访问权限。Unix域套接字使用普通的Unix文件系统权限集。 这个参数值应该是数值的形式，也就是系统调用 `chmod` 和 `umask` 接受的形式。如果使用自定义的八进制格式，数字必须以 `0` 开头。

缺省的权限是 `0777` ， 意思是任何人都可以连接。合理的候选是 `0770` (只有用户和同组的人可以访问， 又见 `unix_socket_group`)和 `0700` (只有用户自己可以访问)。 请注意，对于Unix域套接字，只有写权限有意义， 因此在设置和撤销读和执行权限中没有任何意义。

这个访问控制机制与Chapter 19里描述的用户认证毫无关系。

这个参数只能在服务器启动的时候设置。

这个参数在Windows上是无关紧要的，它不具有Unix域套接字。

`bonjour` (`boolean`)

通过Bonjour启动广播服务器的存在。缺省是关闭的。 这个值只能在服务器启动的时候设置。

`bonjour_name` (`string`)

声明Bonjour服务器名称。 空字符串 `''` (缺省值)表示使用计算机名。 如果编译时没有打开Bonjour支持那么将忽略这个参数。 这个值只能在服务器启动的时候设置。

`tcp_keepalives_idle` (`integer`)

声明发送保持活跃信号的间隔秒数，不发送保持活跃信号，连接就会处于闲置状态。 零使用系统默认值。该参数仅在支持 `TCP_KEEPIDLE` 或者 `TCP_KEEPALIVE` 符号以及 Windows的系统上出现;在其它系统中，它必须是零。 在通过Unix域套接字连接的会话中，这个参数被忽略，并且总是读为零。

> **Note:** 在Windows中，0值将这个参数设置为2小时， 因为Windows不提供读取系统默认值的方法。

`tcp_keepalives_interval` (`integer`)

声明发送保持活跃信号的间隔秒数，不发送保持活跃信号，连接就会处于闲置状态。 零使用系统默认值。 在那些支持 `TCP_KEEPINTVL` 符号以及Windows的系统上支持这些参数。 在其他系统上，这个参数必须为零。 在通过Unix域套接字连接的会话中，这个参数被忽略，并且总是读为零。

> **Note:** 在Windows中，0值将这个参数设置为1秒， 因为Windows不提供读取系统默认值的方法。

`tcp_keepalives_count` (`integer`)

声明发送保持活跃信号的数量，不发送保持活跃信号，连接就会处于闲置状态。 零使用系统默认值。 在那些支持 `TCP_KEEPCNT` 的系统上支持这些参数。 在其他系统上，这个参数必须为零。 在通过Unix域套接字连接的会话中，这个选项被忽略，并且总是读为零。

> **Note:** 在Windows上不支持这个参数，并且总是为零。

18.3.2. 安全和认证

`authentication_timeout` (`integer`)

完成客户端认证的最长时间，以秒计。如果一个客户端没有在这段时间里完成认证协议， 服务器将中断连接。这样就避免了出问题的客户端无限制地占据连接资源。缺省是60秒(`1m`)。 这个选项只能在命令行上或者在 `postgresql.conf` 里设置。

`ssl` (`boolean`)

启用SSL连接。请在使用这个选项之前阅读Section 17.9。 缺省是 `off` 。 这个选项只能在服务器启动的时候设置。SSL通讯与TCP/IP 连接是唯一可能的。

`ssl_ca_file` (`string`)

指定包含SSL服务器证书颁发机构（CA）的文件名称。 默认值为空，意味着没有CA文件被加载， 并且不进行客户端证书验证。 （在以前的PostgreSQL版本中，此文件的名称是作为 `root.crt` 的硬编码。） 相对路径是相对于数据目录。这个参数只能在服务器启动进行设置。

`ssl_cert_file` (`string`)

指定包含SSL服务器证书的文件名称。 默认值是 `server.crt` 。相对路径是 相对于数据目录。 这个参数只能在服务器启动时进行设置。

`ssl_crl_file` (`string`)

指定包含SSL服务器证书撤销列表(CRL)的文件名称。 默认值为空，意味着没有CRL文件被加载。 （在PostgreSQL的早期版本中，该文件的名称为 `root.crl` 的 硬编码。）相对路径是相对于数据目录。这个参数只能在服务器启动时候进行设置。

`ssl_key_file` (`string`)

指定包含SSL服务器私钥的文件名称。 默认值是 `server.key` 。相对路径是 相对于数据目录。这个参数只能在服务器启动时设置。

`ssl_renegotiation_limit` (`integer`)

在会话密钥发生重新商议之前，指定多少数据可以进行SSL-加密。 当大量流量被检查的时候，但它也会产生大量性能损失，重新谈判降低攻击者密码分析的机会 。总的发送和接收的流量用于检查极限。如果该参数被设置为0， 禁用重新谈判。默认值是 `512MB` 。

> **Note:** 当使用SSL时再商议时候，2009年11月之前的SSL库是不安全的，由于SSL协议的一个漏洞。 作为这个漏洞修复，一些厂商出货的SSL库不能重新谈判。 如果在客户端或服务器上使用任何库，应禁用SSL重新协商。

`ssl_ciphers` (`string`)

指定SSL密码列表，在安全连接上允许使用。 参阅openssl手册页支持的密码列表。

`password_encryption` (`boolean`)

在CREATE USER或者ALTER ROLE里声明了一个口令， 而又没有写 `ENCRYPTED` 或者 `UNENCRYPTED` 的时候，这个选项决定口令是否要加密。 缺省是 `on` (加密口令)。

`krb_server_keyfile` (`string`)

设置Kerberos服务器键字文件的位置。参阅Section 19.3.5或者Section 19.3.3获取细节。 这个参数只能在 `postgresql.conf` 文件或者服务器命令行中设置。

`krb_srvname` (`string`)

设置Kerberos服务名。参阅Section 19.3.5获取细节。 这个参数只能在 `postgresql.conf` 文件或者服务器命令行中设置。

`krb_caseins_users` (`boolean`)

设置Kerberos和GSSAPI用户名是否大小写无关。 缺省是 `off` (大小写相关)。 这个参数只能在 `postgresql.conf` 文件或者服务器命令行中设置。

`db_user_namespace` (`boolean`)

这个参数启动每个数据库的用户名。 缺省是关闭的。这个参数只能在命令行上或者在 `postgresql.conf` 里设置。

如果打开这个选项，你应该像 `username@dbname` 这样创建用户。 在给一个正在连接的客户端传递 `username` 的时候，必须给用户名附加 `@` 和数据库名字， 然后服务器查找该数据库相关的用户名字。请注意，如果你在SQL环境里创建包含 `@` 的名字时， 你需要用引号包围用户名。

打开这个选项之后，你还是能够创建普通的全局用户。 只要在客户端声明用户的时候附加一个 `@` 即可。比如 `joe@` 。 在服务器查找这个用户名之前，这个 `@` 会被剥除。

`db_user_namespace` 导致客户端和服务器端的用户名表示不同。 验证检查总是与该服务器的用户名一起进行， 所以验证方法必须配置为服务器的用户名，而不是客户端的。因为 `md5` 在客户端和服务器上使用用户名， `md5` 不能与 `db_user_namespace` 一起使用。

> **Note:** 这个特性只是临时措施，直到找到一个完全的解决方案。那个时候，这个选项将被删除。

18.4. 资源消耗

18.4.1. 内存

`shared_buffers` (`integer`)

设置数据库服务器将使用的共享内存缓冲区数量。缺省通常是128兆字节(`128MB`)， 但是如果你的内核设置不支持这么大，那么可以少些(在initdb的时候决定)。 每个缓冲区大小的典型值是128千字节，（ `BLCKSZ` 的非缺省值改变最小值） 不过，这个数值比最小值大一些通常需要更好的性能。 这个选项只能在服务器启动的时候设置。

如果你有1GB或更多内存的专用数据库服务器， 对于 `shared_buffers` 合理的初始值是您的系统内存的25%。 有一些工作负载，甚至在那里对于 `shared_buffers` 大设置是有效的， 但因为PostgreSQL也依赖于操作系统缓存， 它是不可能的，RAM到 `shared_buffers` 的多于40%的分配比更少数量的工作的更好。 对于 `shared_buffers` 的大量设置 通常要求相应增加 `checkpoint_segments` ， 为了延长写大量新的或者需较长时间修改的数据的进程。

对于少于1GB RAM系统， 较小百分比内存是相应的， 以便为操作系统留有足够的空间。 此外，在Windows上， `shared_buffers` 大点的值不是很有效。 您可能会发现更好的结果保持设置相对较低，并且使用操作系统的缓存代替。 在Windows系统上 `shared_buffers` 的有用范围一般是从64MB到512MB。

`temp_buffers` (`integer`)

设置每个数据库会话使用的临时缓冲区的最大数目。这些都是会话的本地缓冲区， 只用于访问临时表。缺省是8兆字节(`8MB`)。这个设置可以在独立的会话内部设置， 但是只有在会话第一次使用临时表的时候才能增长； 企图在该会话里随后改变该数值是无效的。

一个会话将按照 `temp_buffers` 给出的限制，根据需要分配临时缓冲区。 如果在一个并不需要大量临时缓冲区的会话里设置一个大的数值， 其开销只是一个缓冲区描述符，或者说每个 `temp_buffers` 增加大概64字节。不过， 如果一个缓冲区实际上被使用，那么就会额外消耗8192字节(或者说是 `BLCKSZ` 字节)。

`max_prepared_transactions` (`integer`)

设置可以同时处于"预备"状态的事务的最大数目(参阅PREPARE TRANSACTION)。 把这个参数设置为零（这是缺省值）则关闭预备事务的特性。 这个值只能在服务器启动的时候设置。

如果你不打算使用预备事务，这个参数也可以设置为零。 避免预备事务的偶然建立。如果你使用它们， 你可能会需要把 `max_prepared_transactions` 设置成至少和max_connections 一样大，以避免每个会话可以有预备事务挂起。

当运行备库服务器时，你必须设置相同参数或者比主服务器上更高参数值。 否则，在备库服务器上不允许查询。

`work_mem` (`integer`)

声明内部排序操作和Hash表在开始使用临时磁盘文件之前使用的内存数目。 缺省数值是1兆字节(`1MB`)。请注意对于复杂的查询， 可能会同时并发运行好几个排序或者散列操作；每个都会被批准使用这个参数声明的这么多内存， 然后才会开始求助于临时文件。同样，好几个正在运行的会话可能会同时进行排序操作。 因此使用的总内存可能是 `work_mem` 的好几倍。当选择这个值的时候，必须记住这个事实。 `ORDER BY` , `DISTINCT` 和融合连接都要用到排序操作。 Hash表在散列连接、散列为基础的聚集、散列为基础的 `IN` 子查询处理中都要用到。

`maintenance_work_mem` (`integer`)

声明在维护性操作(比如 `VACUUM` , `CREATE INDEX` 和 `ALTER TABLE ADD FOREIGN KEY`)中使用的最大的内存数。 缺省是16兆字节(`16MB`)。因为在一个数据库会话里， 任意时刻只有一个这样的操作可以执行，并且一个数据库安装通常不会有太多这样的工作并发执行， 把这个数值设置得比 `work_mem` 更大是安全的。 更大的设置可以改进清理和恢复数据库转储的速度。

请注意，当运行自动清理， 直至autovacuum_max_workers次分配这个内存， 所以要小心，不要设置默认值太高。

`max_stack_depth` (`integer`)

声明服务器的执行堆栈的最大安全深度。 为此设置一个参数的原因是内核强制的实际堆栈尺寸(就是 `ulimit -s` 或者局部等效物的设置) 小于安全的一兆字节左右的范围。需要这个安全界限是因为在服务器里，并非所有过程都检查了堆栈深度， 只是在可能递规的过程，比如表达式计算这样的过程里面才进行检查。缺省设置是2兆字节 `2MB` ， 这个值相对比较小，不容易导致崩溃。但是，这个值可能太小了，以至于无法执行复杂的函数。

把 `max_stack_depth` 参数设置得大于实际的内核限制意味着 一个正在运行的递归函数可能会导致一个独立的服务器进程的崩溃。 在PostgreSQL能够检测内核限制的平台上， 服务器将不允许你将其设置为一个不安全的值。 因为并非所有平台都能够检测，所以还是建议你在此设置一个明确的值。

18.4.2. 磁盘

`temp_file_limit` (`integer`)

指定会话可以使用临时文件的最大磁盘空间，如排序和哈希临时文件， 或持有游标的存储文件。一个事务试图超过这个限制将被取消。 该值是指定的千字节，并且 `-1` （缺省）意味着没有限制。 只有超级用户可以更改此设置。

此设置限制任何时刻通过临时文件使用给定PostgreSQL会话使用的总空间。 应当指出的是，使用显式临时表的磁盘空间， 而不是使用查询执行的幕后临时文件， 并强调不影响这个限制。

18.4.3. 内核资源使用

`max_files_per_process` (`integer`)

设置每个服务器进程允许同时打开的最大文件数目。缺省是1000。 如果内核强制一个合理的每进程限制，那么你不用操心这个设置。 但是在一些平台上(特别是大多数BSD系统)， 内核允许独立进程打开比个系统真正可以支持的数目大得多得文件数。 如果你发现有"Too many open files"这样的失败现像，那么就尝试缩小这个设置。 这个值只能在服务器启动的时候设置。

`shared_preload_libraries` (`string`)

这个变量声明一个或者多个在服务器启动的时候预先装载的共享库。 比如 `'$libdir/mylib'` 会在加载标准库目录中的库文件之前预先加载' `mylib.so` (在某些平台上可能是 `mylib.sl`)库文件。 所有库名转换成小写，除非双引号引用。如果有多个库被加载，将他们的名字用逗号分隔。 这个值只能在服务器启动的时候设置。

可以用这个方法预先装载PostgreSQL的过程语言库， 通常是使用 `'$libdir/plXXX'` 语法， 这里的 `XXX` 是 `pgsql` , `perl` , `tcl` 或者 `python` 之一。

通过预先装载一个共享库(以及在需要的时候初始化它)， 我们就可以避免第一次使用这个库的加载时间。不过， 启动每个服务器进程的时间可能会增加， 即使进程从来没有使用过这些库也这样。 因此我们只是建议对那些将被大多数会话使用的库才使用这个选项。

> **Note:** 在Windows主机上，在服务器启动时预加载库并不会减少所需的时间来启动每个新的服务器进程; 每个服务器进程将重新加载所有的预置库。 然而， `shared_preload_libraries` 仍然是在Windows主机上有用的， 因为某些共享库可能需要执行只发生在启动时的某些操作 （例如，一个共享库可能需要预定轻量级锁或共享内存， 启动开始之后你不能这样做）

如果没有找到声明的库，那么服务器启动将失败。

每一个支持PostgreSQL的库都有一个"magic block"用于保证兼容性。 因此，不支持PostgreSQL的库不能用这种办法加载。

18.4.4. 基于开销的清理延迟

在VACUUM和ANALYZE命令执行过程中， 系统维护一个内部的记数器，跟踪所执行的各种I/O操作的近似开销。 如果积累的开销达到了 `vacuum_cost_limit` 声明的限制， 那么执行这个操作的进程将睡眠 `vacuum_cost_delay` 指定的时间。 然后它会重置记数器然后继续执行。

这个特性的目的是允许管理员减少这些命令在并发活动的数据库上的I/O影响。 比如，像 `VACUUM` 和 `ANALYZE` 这样的维护命令并不需要迅速完成， 并且不希望它们严重干扰系统执行其它的数据库操作。 基于开销的清理延迟为管理员提供了一个实现这个目的的手段。

这个特性缺省手动发出 `VACUUM` 命令是关闭的。 要想打开它，把 `vacuum_cost_delay` 变量设置为一个非零值。

`vacuum_cost_delay` (`integer`)

以毫秒计的时间长度，如果超过了开销限制，那么进程将睡眠一会儿。 缺省值0关闭基于开销的清理延迟特性。正数值打开基于开销的清理。 不过，要注意在许多系统上，睡眠的有效分辨率是10毫秒； 把 `vacuum_cost_delay` 设置为一个不是10的整数倍的数值与 将它设置为下一个10的整数倍作用相同。

当使用基于成本的清理， `vacuum_cost_delay` 的适当值通常是相当小的， 也许10或20毫秒。 调节清理的资源消耗最好是通过改变其它清理开销参数完成的。

`vacuum_cost_page_hit` (`integer`)

清理一个在共享缓存里找到的缓冲区的预计开销。 它代表锁住缓冲池、查找共享的Hash表、扫描页面内容的开销。 缺省值是1。

`vacuum_cost_page_miss` (`integer`)

清理一个要从磁盘上读取的缓冲区的预计开销。 它代表锁住缓冲池、查找共享Hash表、从磁盘读取需要的数据块、 扫描它的内容的开销。缺省值是10。

`vacuum_cost_page_dirty` (`integer`)

清理修改一个原先是干净的块的预计开销。 它代表把一个脏的磁盘块再次刷新到磁盘上的额外开销。 缺省值是20。

`vacuum_cost_limit` (`integer`)

导致清理进程休眠的积累开销。缺省是200。

> **Note:** 有些操作会持有关键的锁，并且应该尽快结束。 在这样的操作过程中，基于开销的清理延迟不会发生作用。 因此开销积累远远高于指定的限制是可能的。 为了避免在这种情况下的长延时， 实际的延迟是 `vacuum_cost_delay` *`accumulated_balance`* / *`vacuum_cost_limit`* 与 *`vacuum_cost_delay`* 4 两者之间的最大值。

18.4.5. 后端写进程

从 PostgreSQL 8.0 开始，就有一个独立的服务器进程，叫做后端写进程， 它唯一的功能就是发出写"脏"共享缓冲区的命令。 这么做的目的是让持有用户查询的服务器进程应该很少或者几乎不等待写动作的发生， 因为后端写进程会做这件事情。这样的安排同样也减少了检查点造成的性能下降。 后端写进程将持续的把脏页面刷新到磁盘上，所以在检查点到来的时候，只有几个页面需要刷新到磁盘上。 但是这样还是增加了 I/O 的总净负荷，因为以前的检查点间隔里，一个重复弄脏的页面可能只会冲刷一次， 而同一个间隔里，后端写进程可能会写好几次。在大多数情况下，连续的低负荷要比周期性的尖峰负荷好， 但是在本节讨论的参数可以用于按实际需要调节其行为。

`bgwriter_delay` (`integer`)

声明后端写进程活跃轮回之间的延迟。在每个轮回里， 写进程都会为一些脏的缓冲区发出写操作(可以用下面的参数控制)。 然后它就休眠 `bgwriter_delay` 毫秒，然后重复动作。 当在缓冲池中没有脏缓冲区时，但是，它会无论 `bgwriter_delay` 的值，进入一个较长的睡眠。 缺省值是200(`200ms`)。 请注意在许多系统上，休眠延时的有效分辨率是10毫秒； 因此，把 `bgwriter_delay` 设置为一个不是10的倍数的数值与 设置为下一个10的倍数是一样的效果。 这个选项只能在服务器启动的时候或者在 `postgresql.conf` 文件里设置。

`bgwriter_lru_maxpages` (`integer`)

在每个轮回里， 不超过这么多个缓冲区将通过后端写进程写入磁盘。 设置为零启动后端写进程。（请注意检查点，通过单独的，专用辅助进程来管理，不受影响。） 缺省值是100。这个选项只能在服务器命令行或者在 `postgresql.conf` 文件里设置。

`bgwriter_lru_multiplier` (`floating point`)

写在每一轮的脏缓冲区数目是根据通过最近几轮服务器处理所需的新的缓冲区数。 最近平均需求乘以 `bgwriter_lru_multiplier` 到达将在下一轮中需要的缓冲区的数目的估计。 脏缓冲区写入直到有许多干净的，可重复使用的缓冲区可用。 （但是，每回写入不超过 `bgwriter_lru_maxpages` 的缓冲区。） 因此，1.0的设置表示写入确切预测需要的缓冲区数量的"合适"策略。 较大的值提供针对需求高峰一定的缓冲作用， 而较小的值故意留下服务器进程完成写入。 默认值是2.0。 这个参数只能在 `postgresql.conf` 文件或者服务器命令行中设置。

小的 `bgwriter_lru_maxpages` 和 `bgwriter_lru_multiplier` 减少后端写进程导致的额外I/O负荷， 但是会有可能使服务器进程不得不自己发出写动作，降低查询的交互性。

18.4.6. Asynchronous Behavior

`effective_io_concurrency` (`integer`)

设置PostgreSQL预计可以同时执行的并发磁盘的I/O操作数。 增大该数值将增加任何单独的PostgreSQL会话尝试并行启动的I/O操作数。 允许的范围是1到1000，或者零禁用发出异步I/O请求。目前，此设置只影响堆位图扫描。

这个设置很好的起点是包括一个RAID 0或用于数据库的RAID 1镜像的单独驱动器数。（对于RAID 5奇偶校验驱动器不应该计算在内。） 然而，如果数据库经常忙于在并发会话中发出多个查询， 更低的数值可能是足够的，以保持磁盘阵列繁忙。 比需要保持磁盘繁忙更大的值将只会造成额外的CPU开销。

对于更奇特的系统，如基于内存的存储或由总线带宽限制的RAID阵列，正确的值可能是可用I/O路径数。一些试验可能需要找到最好的值。

异步I/O依赖于有效的`posix_fadvise`功能， 其中一些是操作系统所缺乏的。如果函数不存在，那么这个参数设置为任何东西， 但是零将导致错误。在某些操作系统上（例如Solaris），函数存在，但实际上并没有做任何事情。

18.5. 预写式日志

参阅Section 29.4获取调整这些设置的额外信息。

18.5.1. 设置

`wal_level` (`enum`)

`wal_level` 决定有多少信息被写入到WAL中。 默认值是 `最小的` ，其中写入唯一从崩溃或立即关机中恢复的所需信息。 `archive` 补充WAL归档需要的日志记录，以及 `hot_standby` 进一步增加在备用服务器上运行只读查询所需的信息 。这个参数只能在服务器启动时设置。

在 `最小` 级别中，安全地忽略一些批量操作的WAL-日志， 这可以使那些操作快很多（参阅Section 14.4.7）。 这种优化可以应用到：

`CREATE TABLE AS`
`CREATE INDEX`
`CLUSTER`
`COPY` into tables that were created or truncated in the same transaction

但是最小的WAL不包含从基本的备份和WAL日志中重建数据的足够信息， 所以 `archive` 或者 `hot_standby` 级别必须用来启动 WAL归档(archive_mode)和流复制。

在 `hot_standby` 级别，相同的信息被记录为 `archive` ， 加上需要重建从WAL运行的事务状态信息。 为了在备用服务器上启用只读查询，主库上 `wal_level` 必须设置为 `hot_standby` ， 必须启动备库上的hot_standby。 使用 `hot_standby` 和 `archive` 级别之间的性能几乎没有可测量的差异， 如果任何生产的影响是明显的，则是值得欢迎的。

`fsync` (`boolean`)

如果打开这个选项，那么PostgreSQL服务器将在好几个地方使用 `fsync()` 系统调用 (或等价调用，参见wal_sync_method)来确保更新已经物理上写到磁盘中。 这样就保证了数据库集群将在操作系统或者硬件崩溃的情况下恢复到一个一致的状态。

当关闭 `fsync` 时通常是性能利益， 这可能会导致发生断电或系统崩溃时不可恢复数据丢失。 如果你可以很容易的从外部数据中创建您的整个数据库， 因此关闭 `fsync` 是明智的。

关闭 `fsync` 安全情况的例子包括备份文件中新的数据库集群的初始加载， 数据库被丢弃和重新创建之后，使用数据库集群批处理数据， 或者为只读数据库克隆被频繁重建并且不用于故障转移。 为了关闭 `fsync` 高质量硬件本身是不充分的。

当改变 `fsync` 从关闭到打开时，对于可靠的恢复， 必须强制内核中所有被修改的缓冲区到持久存储。 当集群宕机或当fsync通过运行 `initdb--sync-only` ， `sync` ， 卸载文件系统，或重新启动服务器的时候，再完成这项工作。

在许多情况下，关闭synchronous_commit 为非关键的事务可以提供关闭 `fsync` 的潜力性能优势， 没有数据损坏随之而来的风险。

`fsync` 只能在postgresql.conf文件里或者服务器命令行里设置。 如果这个参数被关闭，那么请考虑把full_page_writes也关闭了。

`synchronous_commit` (`enum`)

命令返回"成功"指示给客户端之前，指定是否事务提交将等待WAL记录被写入到磁盘。 有效值是 `on` , `remote_write` , `local` 和 `off` 。 默认情况下，安全设置是 `on` 。 当 `off` 时，当成功报告给客户端， 并当该事务真正保证是安全的，不会在服务器崩溃的时候，可以有一定的延时（最大 延迟wal_writer_delay的3倍）。 不同于fsync，将此参数设置为 `off` 不会产生任何数据库不一致的风险： 操作系统或数据库崩溃可能导致丢失一些最近提交的事务， 但数据库状态将是一样的，正如该事务已经彻底终止。 因此，当性能比准确事务耐久性更重要时，关闭 `synchronous_commit` 是有效选择。 获取更多讨论请参阅Section 29.3。

如果设置synchronous_standby_names， 该参数控制是否事务提交将等待它的WAL记录被复制到备用服务器。 当设置 `on` 的时候， 提交将等待直到回复当前同步备库表明它已收到事务提交记录，并刷新到磁盘。 这确保事务不会丢失，除非主库和备库遭受他们的数据库存储崩溃。 当设置为 `remote_write` ，事务将等待 直到当前同步备库的答复表明它已经收到事务的提交记录，并且写入到备用操作系统， 但是数据并不一定在备库中达到稳定存储。 即使PostgreSQL备库实例崩溃，但并非备库遭受操作系统级的崩溃，此设置足以 确保数据的保存。

当同步复制使用时， 通常是明智的，要么等待本地刷新到磁盘和WAL记录的复制，要么 允许异步提交事务。然而，该设置 `local` 可用于 希望等待本地刷新到磁盘上的事务，而不是同步复制。 如果不设置 `synchronous_standby_names` ， 设置 `on` , `remote_write` 和 `local` 提供相同的同步级别：事务提交只能等待本地刷新到磁盘。

该参数可以在任何时候被改变;对于任何事务的行为 是由该设置提交生效时确定的。 因此，它是可能的，并且有用的， 有一些事务同步提交，其他的异步提交。 例如，为了使单一多语句事务异步提交，缺省是相反的， 在事务中发出 `SET LOCAL synchronous_commit TO OFF` 命令。

`wal_sync_method` (`enum`)

用来向磁盘强制更新WAL数据的方法。如果 `fsync` 是关闭的， 那么这个设置就没有意义，因为所有WAL文件更新都不会强制输出。 可能的值是：

- `open_datasync` (用带 `O_DSYNC` 选项的 `open()` 打开WAL文件)
- `fdatasync` (每次提交的时候都调用 `fdatasync()`)

- `fsync` (每次提交的时候都调用 `fsync()`)
- `fsync_writethrough` (每次提交的时候调用 `fsync()` 强制写出任何磁盘写缓冲区)
- `open_sync` (用带 `O_SYNC` 选项的 `open()` 写WAL文件)

如果可用的话，该 `open_` *选项也使用 `O_DIRECT` 。 并非所有这些选择在所有平台上都可用。默认值是平台支持的上面列表中的第一个方法， 除了 `fdatasync` 在Linux上是缺省的。 缺省的也不一定理想; 为了创建安全配置或达到最佳性能， 可能要改变这个设置或你的系统配置的其他方面， 将在Section 29.1中讨论。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

`full_page_writes` (`boolean`)

打开这个选项的时候，PostgreSQL服务器在检查点之后对页面的第一次写入时将整个页面写到 WAL 里面。 这么做是因为在操作系统崩溃过程中可能只有部分页面写入磁盘， 从而导致在同一个页面中包含新旧数据的混合。在崩溃后的恢复期间， 由于在WAL里面存储的行变化信息不够完整，因此无法完全恢复该页。 把完整的页面影像保存下来就可以保证正确存储页面， 代价是增加了写入WAL的数据量。因为WAL重放总是从一个检查点开始的， 所以在检查点后每个页面第一次改变的时候做WAL备份就足够了。 因此，一个减小全页面写开销的方法是增加检查点的间隔参数值。

把这个选项关闭会加快正常操作的速度， 但是可能导致系统崩溃后不可恢复的数据损坏或者无记载数据损坏， 它的危害类似于 `fsync` ， 只是比较小而已。并且在建议参数相同的情况下关闭这个选项。

关闭这个选项并不影响即时恢复(PITR)的WAL使用(参阅Section 24.3)。

这个选项只能在 `postgresql.conf` 文件里或者服务器命令行设置。缺省是 `on` 。

`wal_buffers` (`integer`)

使用已经写入磁盘的WAL数据共享内存的数量。-1的默认设置选择大小等于 shared_buffers 的1/32nd(大约3%)， 但不小于 `64kB` 也不超过一个WAL段大小， 通常 `16MB` 。如果自动选择过大或过小，则可以手动设置这个值。 但任何小于 `32kB` 的正值将 当作 `32kB` 处理。 这个参数只能在服务器启动时设置。

WAL缓冲区的内容每次事务提交时写入到磁盘， 这样非常大的值不可能提供显著的好处。 但是，当有许多客户端都同时提交时，设置该值至少为几兆可以提高繁忙服务器的写入性能，在多数情况下，由-1的缺省设置选择自动调整应给予合理的结果。

`wal_writer_delay` (`integer`)

声明WAL写入进程的活动轮回的延迟。在每一轮回中写入进程将刷新WAL到磁盘。 然后，它会休眠 `wal_writer_delay` 毫秒，并重复。 默认值是200毫秒(`200ms`)。注意， 在许多系统上，睡眠延迟的有效分辨率为10毫秒; 把 `wal_writer_delay` 的值 设置为一个不是10的倍数的

数值与设置为下一个10的倍数是一样的效果。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

`commit_delay` (`integer`)

`commit_delay` 增加了时间延迟，在WAL刷新启动之前，以微秒测量。 这可以提高通过单一WAL刷新提交大量事务的组提交吞吐量。 如果系统负载足够高，额外事务在给定时间间隔内成为提交。 然而，它也增加了每个WAL刷新的最多 `commit_delay` 微秒的延迟时间。 但是如果没有其它事务准备提交，那么这个间隔就是在浪费时间。 如果至少 `commit_siblings` 个其它事务是活跃的，当刷新初始化的情况下。 则仅仅只执行一个延迟。 如果禁用 `fsync` ， 则不执行任何延迟。 缺省 `commit_delay` 是零（无延迟）。 只有超级用户可以更改此设置。

在PostgreSQL先于9.3发布的版本中， `commit_delay` 表现不同，更别说效能： 它仅影响提交，而不是所有的WAL刷新， 即使WAL刷新尽早完成了，也要等待整个配置延迟。 在PostgreSQL9.3开始， 第一个过程准备刷新等待配置延迟，而随后的过程仅仅等待直到完成刷新操作。

`commit_siblings` (`integer`)

在执行 `commit_delay` 延迟的时候，要求最少同时打开的并发事务数目。 大一些的数值会导致在延迟期间另外一个事务准备好提交的可能性增大。 缺省是5。

18.5.2. 检查点

`checkpoint_segments` (`integer`)

在自动的WAL检查点之间的日志文件段的最大数量(通常每个段16兆字节)。 缺省是3。增加这个参数可以增加崩溃恢复所需要的时间量。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。

`checkpoint_timeout` (`integer`)

在自动WAL检查点之间的最长时间，以秒计。缺省是5分钟(`5min`)。 增加这个参数可以增加崩溃恢复所需要的时间量。 这个选项只能在 `postgresql.conf` 文件或者服务器命令行中设置。

`checkpoint_completion_target` (`floating point`)

声明检查点完成的目标，作为检查点之间总时间的分数。缺省是0.5。 这个参数只能在 `postgresql.conf` 文件中或者服务器命令行中设置。

`checkpoint_warning` (`integer`)

如果由于填充检查点段文件导致的检查点发生时间间隔接近这个参数表示的秒数， 那么就向服务器日志发送一个建议增加 `checkpoint_segments` 值的消息。 缺省是30秒(`30s`)。零则关闭警告。 如果 `checkpoint_timeout` 小于 `checkpoint_warning` ， 则不产生警告。这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。

18.5.3. 归档

`archive_mode` (`boolean`)

当启用 `archive_mode` 的时候， 已完成的WAL段通过设置archive_command 发送到归档存储。 `archive_mode` 和 `archive_command` 是独立变量， 以致于 `archive_command` 不留归档模式而被改变。 这个参数只能在服务器启动时设置。 当 `wal_level` 设置为 `minimal` 时，则不启用 `archive_mode` 。

`archive_command` (`string`)

将一个完整的WAL文件序列归档的shell命令。 字符串中任何 `%p` 都被要归档的文件的绝对路径代替， 而任何 `%f` 都只被该文件名代替(非绝对路径都相对于集群的数据目录)。 如果你需要在命令里嵌入 `%` 字符就必须双写 `%%` 。 有一点很重要：这个命令必须是当且仅当成功的时候才返回零。 参阅Section 24.3.1获取更多的信息。

这个参数只能在 `postgresql.conf` 文件或服务器命令行上。 除非在服务器启动时开启 `archive_mode` ，否则忽略它。 如果 `archive_command` 是一个空字符串（缺省）而 `archive_mode` 已启用，暂时禁用WAL归档， 但是服务器仍继续堆积WAL段文件期望迅速提供一个命令。 设置 `archive_command` 命令什么也不做但返回true `/bin/true` (Windows上的 `REM`)，有效地禁用归档， 但也为了归档恢复打破了所需WAL文件链，所以应该只在特殊情况下使用。

`archive_timeout` (`integer`)

archive_command仅在已完成的WAL段上调用。 因此，如果服务器只产生很少的WAL流量(或产生流量的周期很长)， 那么在完成事务以及安全归档存储之间将有一个很长的延时。 为了限制未归档数据的逗留时间， 你可以强制服务器以 `archive_timeout` 指定的秒数为周期切换到新的WAL段文件。 当这个参数大于零时，服务器将切换到新的段文件，无论从剩余段文件切换过去多少秒， 并且有任何的数据库活动，包含一个单独的检查点（增加 `checkpoint_timeout` 将减少空闲系统上不必要的检查）。 注意，由于强制切换而提早关闭的归档文件仍然与完整的归档文件长度相同。 因此，将 `archive_timeout` —设为很小的值是不明智的，它将导致占用巨大的归档存储空间。 将 `archive_timeout` 设置为60秒左右是比较合理的。 如果你想要拷贝主服务器数据更加快速，你应该考虑使用流复制，而不是归档。 这个选项只能在 `postgresql.conf` 文件或者服务器命令行里设置。

18.6. 复制

这些设置控制流复制特性(参见 Section 25.2.5)。 服务器将是任何一个主或备用服务器。主服务器可以发送数据， 而备用(s)总是拷贝数据的接收器。当使用级联复制时(参见Section 25.2.6)， 备用服务器(s)也可以是发送者，也可以是接收器。 参数主要用于发送和备用服务器， 虽然有些参数仅在主服务器上。 如果是必需的，那么通过集群设置可能会有所不同。

18.6.1. 发送服务器

这些参数可以在发送复制的数据给一个或多个备用服务器的任何服务器上进行设置。 主服务器总是发送服务器，所以这些参数必须总是在主服务器上设置。 备用成为主之后这些参数的作用和意义不会改变。

`max_wal_senders` (`integer`)

指定来自备用服务器或流基础备份客户端的并发连接的最大数目 （即同时运行WAL发送者进程的最大数目）。 默认值是零，这意味着禁用复制。 WAL发送者进程计算连接总数， 因此参数不能高于max_connections。 这个参数只能在服务器启动时设置。 `wal_level` 必须设置为 `archive` 或者 `hot_standby` 允许来自备用服务器的连接。

`wal_keep_segments` (`integer`)

指定在 `pg_xlog` 目录下的以往日志文件段的最小数量， 如果备用服务器为了流复制需要获取它们。 那么每个段通常是16兆字节。 如果备用服务器连接到发送服务器落后于 `wal_keep_segments` 段，那么发送服务器可能会删除WAL段仍需要待机状态，在这种情况下， 复制连接将被终止。下游连接也将最终失败，因为其结果。 （但是，备用服务器可以从归档文件读取的段进行恢复，如果WAL归档在使用中。）

设置保留在 `pg_xlog` 中的段最小数量; 该系统可能需要为WAL归档或从检查点恢复保留更多段。 如果 `wal_keep_segments` 为0（默认）， 系统不保留备用目的的任何额外段， 所以提供给备用服务器的旧WAL段数是以前检查点定位函数和WAL归档状态信息。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

`wal_sender_timeout` (`integer`)

终止比指定毫秒数闲置更长时间的复制连接。 这对于发送服务器检测待机死机或网络中断是很有帮助的。 零值将禁用超时机制。 此参数只能在 `postgresql.conf` 文件或服务器命令行上设置。 默认值是60秒。

18.6.2. 主服务器

这些参数可以在主/首要的服务器上设置，它将复制的数据发送给一个或多个备用服务器。 需要注意的是，除了这些参数，wal_level必须在主服务器上适当地设定， 并且启用任选的WAL归档（参见Section 18.5.3）。 备用服务器上这些参数值是不相关的，虽然你可能希望设置它们，为了备用的成为主服务器可能性做准备。

`synchronous_standby_names` (`string`)

指定用逗号分隔的备用名称列表，可以支持同步复制， 如Section 25.2.7描述的。 任何一个时间将至多有一个活跃同步备用; 这个备用服务器确认收到他们的数据后，等待提交事务将被允许进行。 同步待机是此列表中第一个备用。列表是当前连接和实时数据流（如通过 `pg_stat_replication` 视图中的 `streaming` 状态显示）。 之后出现在此列表中的其它备用服务器带来潜在的同步备用。 如果无论出于何种原因当前同步待机断开，那么它会立即被下一个最高优先级的替换。 指定多个备用名可以有非常高的可用性。

备用服务器用于此目的的名称是备用 `application_name` 的设定， 正如备用walreceiver中 `primary_conninfo` 的设置。 没有任何机制来保证唯一性。重复匹配备用记录中的一个的情况下将被选作同步待机， 虽然哪一个是不确定的。 特殊项 `*` 匹配任何的 `application_name` ，包括 `walreceiver` 的默认应用程序名称。

如果没有在这里指定同步备用名，那么不启用同步复制并且事务提交将不会等待复制。这是默认配置。 即使当已经启用同步复制，个别事务可以配置而不等待复制，它通过设置synchronous_commit参数到 `local` 或者 `off` 。

这个参数只能在 `postgresql.conf` 文件或者服务器命令行设置。

`vacuum_defer_cleanup_age` (`integer`)

指定由 `VACUUM` 和 HOT更新的事务数将延迟死行版本的清理。 该默认值是零事务，这意味着死行版本可以尽快删除， 也就是说，只要他们不再可见于任何打开的事务。 你不妨将它设置为一个支持双机热备的主服务器上的非零值， 正如Section 25.5所描述的。 这需要更多的时间完成待机状态的查询，由于行早期清除而不会产生冲突。 然而，因为该值是依据发生在主服务器上的写入事务数量进行计算， 它是很难预测到底有多少额外的宽限时间将提供给备用查询。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

你也应该考虑在备用服务器(s)设置 `hot_standby_feedback` ，作为使用该参数的另外一种选择。

18.6.3. 备用服务器

这些设置控制备用服务器的行为以接收复制数据。 在主服务器上的值是不相关的。

`hot_standby` (`boolean`)

指定恢复期间是否可以连接并运行查询，如在Section 25.5中所描述的。 默认值是 `off` 。 这个参数只能在服务器启动时设置。它在存档恢复或处于待机模式时见效。

`max_standby_archive_delay` (`integer`)

当热备是活跃时，这个参数决定取消与应用的WAL项冲突的备用查询之前，备用服务器应等待多久。 如Section 25.5.2所描述的。 `max_standby_archive_delay` 适用于WAL数据从WAL归档读取（因此不是现在）。 默认值是30秒。如果不指定，则单位是毫秒。 值为-1允许待机永远等待完成查询冲突。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

注意， `max_standby_archive_delay` 和 运行查询取消之前的时间最大长度不一样; 而是可以申请任何一个WAL段数据的最大总时间。 因此，如果一个查询导致WAL段中早期显著延迟， 随后的冲突查询将有少得多的时间。

`max_standby_streaming_delay` (`integer`)

当热备是活跃时，这个参数决定取消与应用的WAL项冲突的备用查询之前，备用服务器应等待多久。 如Section 25.5.2所描述的。 `max_standby_streaming_delay` 适用于WAL数据从流复制接收到。 默认值是30秒。如果不指定，则单位是毫秒。 值为-1允许待机永远等待完成查询冲突。 这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。

注意， `max_standby_streaming_delay` 和 运行查询取消之前的时间最大长度不一样; 一旦已经从主服务器接收，则是可以申请任何一个WAL段数据的最大总时间。 因此，如果一个查询导致显著延迟， 随后的冲突查询将有少得多的时间直到备用服务器再次追赶上。

`wal_receiver_status_interval` (`integer`)

指定WAL接收的最小频率，处理备库上发送有关复制进程信息到主或上游待机状态， 在那里可以使用可见的 `pg_stat_replication` 视图。 待机会报告它已写入的最后一个事务日志的位置， 最后一个位置已经刷新到磁盘中，并已申请最后位置。 此参数值的最大时间间隔，以秒为单位。 更新每次写或刷新的位置变化，或者至少往往由这个参数所指定。因此， 应用位置可能稍微落后于真实的位置。此参数设置为零完全禁用状态更新。这个参数只能在 `postgresql.conf` 文件或服务器命令行上设置。默认值是10秒。

`hot_standby_feedback` (`boolean`)

指定热备是否将发送反馈到主或有关查询当前正在备机上执行的上游备机。此参数可以 用于消除查询取消引起清除的记录，但可能会导致主机某些工作负载的数据库膨胀。 反馈信息将不会被更频繁地发送超过一次 `wal_receiver_status_interval` 。缺省值是 `关闭的` 。这个参数只能在设置 `postgresql.conf` 文件或服务器命令行上。

如果级联复制是使用中的反馈通过上游直到它最终到达主机。备用不作任何其他用途 反馈他们收到以外的其他上游传递。

`wal_receiver_timeout` (`integer`)

终止比指定毫秒数闲置更长时间的复制连接。 这对于发送服务器检测待机死机或网络中断是很有帮助的。 零值将禁用超时机制。 此参数只能在 `postgresql.conf` 文件或服务器命令行上设置。 默认值是60秒。

18.7. 查询规划

18.7.1. 规划器方法配置

这些配置参数提供了影响查询优化器选择查询规划的原始方法。 如果优化器为特定的查询选择的缺省规划并不是最优， 那么我们就可以通过使用这些配置参数强制优化器选择一个不同的规划来临时解决这个问题。 更好的改善优化器选择规划的方法 包括调节规划器开销常量、手动运行ANALYZE、增大配置参数default_statistics_target的值、 使用 `ALTER TABLE SET STATISTICS` 为某个字段增加收集的统计信息。

`enable_bitmapscan` (`boolean`)

打开或者关闭规划器对位图扫描规划类型的使用。缺省是 `on` 。

`enable_hashagg` (`boolean`)

打开或者关闭规划器对Hash聚集规划类型的使用。缺省是 `on` 。

`enable_hashjoin` (`boolean`)

打开或者关闭规划器对Hash连接规划类型的使用。缺省是 `on` 。

`enable_indexscan` (`boolean`)

打开或者关闭规划器对索引扫描规划类型的使用。缺省是 `on` 。

`enable_indexonlyscan` (`boolean`)

打开或关闭规划器对唯一索引扫描规划类型的使用。缺省是 `on` 。

`enable_material` (`boolean`)

打开或关闭查询规划器使用物化。不可能完全抑制物化，但是 关闭这个变量会阻止规划器插入物化节点，除非它是必需正确的情况。 默认值是 `on` 。

`enable_mergejoin` (`boolean`)

打开或者关闭规划器对融合连接规划类型的使用。缺省是 `on` 。

`enable_nestloop` (`boolean`)

打开或者关闭规划器对嵌套循环连接规划类型的使用。 我们不可能完全消除嵌套循环连接，但是把这个变量关闭就会让规划器在存在其它方法的时候优先选择其它方法。 缺省是 `on` 。

`enable_seqscan` (`boolean`)

打开或者关闭规划器对顺序扫描规划类型的使用。我们不可能完全消除顺序扫描， 但是把这个变量关闭会让规划器在存在其它方法的时候优先选择其它方法。 缺省是 `on` 。

`enable_sort` (`boolean`)

打开或者关闭规划器使用明确的排序步骤。我们不可能完全消除明确的排序， 但是把这个变量关闭可以让规划器在存在其它方法的时候优先选择其它方法。 缺省是 `on` 。

`enable_tidscan` (`boolean`)

打开或者关闭规划器对TID扫描规划类型的使用。缺省是 `on` 。

18.7.2. 规划器开销常量

本节中描述的开销可以按照任意标准度量。我们只关心其相对值， 因此以相同的系数缩放它们将不会对规划器产生任何影响。传统上， 它们以抓取顺序页的开销作为基准单位。也就是说将 `seq_page_cost` 设为 `1.0` ， 同时其它开销参数对照它来设置。当然你也可以使用其它基准， 比如以毫秒计的实际执行时间。

> **Note:** 糟糕的是，现在还没有定义得很合理的方法来判断下面出现的"开销"变量族的理想数值。 它们最好按照某个特定安装的平均查询开销来衡量。 这意味着仅仅根据很少量的试验结果来修改它们是很危险的。

`seq_page_cost` (`floating point`)

设置规划器计算一次顺序磁盘页面抓取的开销。默认值是1.0。 这个值可以通过设置同名表空间参数表的特定表和索引覆盖。 （参阅ALTER TABLESPACE）

`random_page_cost` (`floating point`)

设置规划器计算一次非顺序磁盘页面抓取的开销。默认值是4.0。 这个值可以通过设置同名表空间参数表的特定表和索引覆盖。 （参阅ALTER TABLESPACE）。

(相对于 `seq_page_cost`)减少这个值将导致更倾向于使用索引扫描， 而增加这个值将导致更倾向于使用顺序扫描。 可以通过同时增加或减少这两个值来调整磁盘I/O相对于CPU的开销(在下面的参数中描述)。

机械磁盘存储的随机访问比4次顺序访问往往更加昂贵。然而，下面使用缺省（4.0）， 因为大多数随机访问磁盘，比如索引读取是在缓存中。 默认值可以作为模拟随机存取比顺序存取慢40倍，而预计90%随机读取到缓存中。

如果您认为90%的缓存率是你的工作量的错误假设， 你可以增加random_page_cost更好地反映随机存储读取的真实成本。相应地， 如果你的数据很可能完全在高速缓存中，如当 数据库比总的服务器内存较小的时候，可以适当减少 random_page_cost。存储具有相对顺序的低随机读取成本， 例如固态硬盘，可能还可以更好地与random_page_cost的低值建模。

> **Tip:** 虽然允许你将`random_page_cost`设置的比`seq_page_cost`小， 但是物理上的实际情况并不受此影响。然而当所有数据库都位于内存中时， 两者设置为相等是非常合理的，因为在此情况下，乱序抓取并不比顺序抓取开销更大。同样， 在缓冲率很高的数据库上，你应当相对于CPU开销同时降低这两个值， 因为获取内存中的页比通常情况下的开销小许多。

`cpu_tuple_cost` (`floating point`)

设置规划器计算在一次查询中处理一个数据行的开销。缺省是0.01。

`cpu_index_tuple_cost` (`floating point`)

设置规划器计算在一次索引扫描中处理每条索引行的开销。缺省是0.005。

`cpu_operator_cost` (`floating point`)

设置规划器计算在一次查询中执行一个操作符或函数的开销。 缺省是0.0025。

`effective_cache_size` (`integer`)

为规划器设置在一次索引扫描中可用的磁盘缓冲区的有效大小。 这个参数会在计算一个索引的预计开销值的时候加以考虑。更高的数值会导致更可能使用索引扫描， 更低的数值会导致更有可能选择顺序扫描。在设置这个参数的时候， 你还应该考虑PostgreSQL的数据文件会使用的共享缓冲区和内核的磁盘缓冲区。另外， 还要考虑预计会使用不同索引的并发查询数目，因为它们必须共享可用的内存空间。 这个参数对PostgreSQL分配的共享内存大小没有影响，它也不会使用内核磁盘缓冲， 它只用于估算。该系统还并未假设数据仍保留在查询之间的磁盘缓存中。默认值是128兆字节(`128MB`)。

18.7.3. 基因查询优化器

基因查询优化（GEQO）是一种算法，采用启发式搜索查询规划。这样可以为复杂查询（链接着很多关系）减少规划时间， 生产规划成本有时低于由正常穷举搜索算法发现的那些。 获取更多信息，请参阅Chapter 53。

`geqo` (`boolean`)

允许或禁止基因查询优化，这是缺省值。 在生产中最好不要关闭它。 `geqo_threshold` 变量提供了GEQO的更精细方法。

`geqo_threshold` (`integer`)

只有当涉及的 `FROM` 关系数量至少有这么多个的时候，才使用基因查询优化。 （请注意一个 `FULL OUTER JOIN` 构造只算一个 `FROM` 项）。缺省是12。 对于数量小于此值的查询，也许使用判定性的穷举搜索更有效。但是对于有许多表的查询， 规划器做判断要花很多时间。往往比执行一个次优规划时间更长。 因此，查询大小阈值是管理使用GEQO的简单方法。

`geqo_effort` (`integer`)

控制GEQO里规划时间和查询规划的有效性之间的平衡。这个变量必须是一个范围从1到10的整数。 缺省值是5。大的数值增加花在进行查询规划上面的时间， 但是也很可能会提高选中更有效的查询规划的几率。

`geqo_effort` 实际上并没有直接干什么事情； 只是用于计算其它那些影响GEQO行为变量的缺省值(在下面描述)。 如果你愿意，你可以手工设置其它参数。

`geqo_pool_size` (`integer`)

控制GEQO使用的池大小。池大小是基因全体中的个体数量。它必须至少是2， 并且有用的数值通常在100和1000之间。如果把它设置为零(缺省)， 那么就会基于 `geqo_effort` 和查询中表的数量选取一个合适的值。

`geqo_generations` (`integer`)

控制GEQO使用的子代数目。子代的意思是算法的迭代次数。它必须至少是1， 有用的值范围和池大小相同。如果设置为零(缺省)，那么将基于 `geqo_pool_size` 选取合适的值。

`geqo_selection_bias` (`floating point`)

控制GEQO使用的选择性偏好。选择性偏好是在一个种群中的选择性压力。数值可以是1.5到2.0之间； 缺省是2.0。

`geqo_seed` (`floating point`)

控制随机数发生器的初始值，它使用由GEQO通过连接顺序搜索空间来选择随机路径。 该值的范围可以从零（默认值）到一。 各种不同的值改变探索连接路径的设置， 并可能导致发现或好或坏的最佳路径。

18.7.4. 其它规划器选项

`default_statistics_target` (`integer`)

为没有用 `ALTER TABLE SET STATISTICS` 设置字段相关目标的表中其它字段设置缺省统计目标。更大的数值增加了 `ANALYZE` 所需要的时间，但是可能会改善规划器的估计质量。 缺省值是100。 有关PostgreSQL的查询规划器使用的统计的更多信息，请参考Section 14.2。

`constraint_exclusion` (`enum`)

控制查询规划器使用的表约束优化查询。 `constraint_exclusion` 允许的值是 `on` （检查所有表的约束）， `off` （永远不检查约束），以及 `partition` （检查仅用于继承子表和 `UNION ALL` 子查询的约束）。 `partition` 是默认设置。它往往使用继承和分区表来提高性能。

当这个参数为特定表允许时，那么规划器用查询条件和 `CHECK` 约束进行比较， 并且在查询条件和约束冲突的情况下，忽略对表的扫描。比如：

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

在打开约束排除的时候，这个 `SELECT` 将完全不会扫描 `child1000` 。 这样可以提高性能。

目前，constraint exclusion默认启用， 仅适用于那些通常用来实现表分区情况。 在简单查询中为所有表强加额外开销计划而打开它是很明显的， 并且经常会产生不受益于简单查询。 如果你没有分区表，您可能更愿意将其完全关闭。

参考Section 5.9.4获取有关使用约束排除和分区的更多信息。

`cursor_tuple_fraction` (`floating point`)

设置被检索的游标行分数的规划器估计。 默认值是0.1。这个设置较小的值 偏好规划器使用"fast start"规划游标， 当可能花费很长时间读取所有行时，这将很快检索出前几行 。较大的值把更多的重点放在总的估计时间上。 1.0的最大设置，规划游标类似于定期查询， 只考虑总估计时间，而不是多长时间传递第一行。

`from_collapse_limit` (`integer`)

如果生成的 `FROM` 列表不超过这个限制的项数，规划器将把子查询融合到上层查询。 小的数值降低规划的时间，但是可能会生成差些的查询计划。缺省是8。 更多信息请查看Section 14.3。

设置这个值到geqo_threshold或者GEQO规划器触发， 导致非最优规划。参阅Section 18.7.3。

`join_collapse_limit` (`integer`)

如果得出的列表不超过这个数目的项， 那么规划器将把除 `FULL JOIN` 之外的 `JOIN` 构造抹平到 `FROM` 列表项中。 小的数值降低规划的时间，但是可能会生成差些的查询计划。

缺省时，这个值和 `from_collapse_limit` 相同，这样适合大多数场合。 把它设置为1则避免任何 `JOIN` 的融合，这样就将明确使用语句中的连接顺序。 查询优化器并不是总能选取最优的连接顺序；高级用户可以选择暂时把这个变量设置为1， 然后明确地声明他们需要的连接顺序。更多信息参见Section 14.3。

设置这个值到geqo_threshold或者GEQO规划器触发， 导致非最优规划。参阅Section 18.7.3。

18.8. 错误报告和日志

18.8.1. 在哪里记录日志

`log_destination` (`string`)

PostgreSQL支持多种记录服务器日志的方法，包括stderr, csvlog和 syslog。 在Windows里，还支持eventlog。把这个选项设置为一个逗号分隔的日志目标的列表。 缺省是只记录到stderr。这个选项只能在 `postgresql.conf` 文件 或者服务器命令行设置。

如果csvlog包含在 `log_destination` 中， 日志项是用"逗号分隔"(CSV)格式的输出，这便于加载日志到程序。参见Section 18.8.4获取更多详情。 logging_collector必须能够产生CSV格式的日志输出。

> **Note:** 在大多数Unix系统上，你将需要改变您的系统syslog守护进程的配置， 以便充分利用 `log_destination` 的syslog选项。 PostgreSQL可以通过 `LOCAL7` (参见syslog_facility)记录syslog设施 `LOCAL0`， 但缺省大多数平台上syslog配置将忽略所有这样的消息。 你将需要添加类似于下面的信息：
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> 到syslog守护程序的配置文件中，使其工作。
>
> 在Windows上，当您使用 `log_destination` 的 `eventlog` 选项， 你应该注册一个事件源及其操作系统作业库， 使Windows事件查看器可以有规则的显示事件日志信息。 参见Section 17.11获取更多详细信息。

`logging_collector` (`boolean`)

这个参数启动日志收集， 这是一个后台进程，抓取发送到stderr的日志消息，并会将他们重定向到日志文件。 这个方法通常比记录到syslog更有用， 因为有些消息类型可能不会出现在syslog输出中 (一个常见的例子是动态连接失败的消息；另外一个是通过脚本比如 `archive_command` 产生的错误消息。） 这个值只能在服务器启动的时候设置。

> **Note:** 不使用日志收集器可以登录到stderr; 无论服务器的stderr被定向到哪， 则日志消息就定位到那里。然而， 该方法是只适用于低日志卷，因为它没有提供方便的方式来旋转日志文件。 另外，在某些平台上不使用日志收集器可能会导致丢失或乱码日志输出， 因为多个进程同时写入同一个日志文件可能覆盖彼此的输出。
>
> **Note:** 日志收集器设计永远不会丢失消息。 这意味着在非常高的负载情况下， 当收集器已经落后而试图发送额外的日志消息的时候，封锁服务器进程。 与此相反，syslog更喜欢忽略消息， 如果它无法写入，这意味着它可能无法记录这些消息，这种情况下， 但它不会阻止该系统的其余部分。

`log_directory` (`string`)

在打开了 `logging_collector` 的时候，这个选项判断日志文件在哪个目录里创建。 它可以声明成绝对路径，或者是与集群的数据目录相对的路径。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行设置。

`log_filename` (`string`)

在打开了 `logging_collector` 的时候，这个选项设置所创建的日志文件的文件名。 这个数值将被当作 `strftime` 模式看待。因此可以用 `%` 逃逸声明随时间而变的文件名。 （注意，如果有任何时区相关 `%` 逃逸，由log_timezone声明的时区进行计算。） 支持的 `%` 逃逸类似于Open Group上的strftime规范列表中的。 注意，不直接使用这个系统的strftime，所以特定平台的（非标准）扩展不起作用。

如果您没有逃逸指定文件名，你应该计划使用日志旋转程序以避免最终填充 整个磁盘。在8.4之前的版本中，如果没有 `%` 的转义符出现， PostgreSQL将追加新日志文件创建时间的时间戳， 但是不再是这种情况了。

如果启用 `log_destination` 中的CSV格式输出， `.csv` 将追加时间戳日志文件名来创建CSV格式输出的文件名。 （如果 `log_filename` 以 `.log` 为结束，后缀代替）。 在上面的例子情况下，CSV文件名是 `server_log.1093827753.csv` 。

这个参数只能在 `postgresql.conf` 文件里或者服务器命令行上设置。

`log_file_mode` (`integer`)

在Unix系统上，当 `logging_collector` 已启用时（在Microsoft Windows上将忽略此参数）， 此参数用于设置日志文件的权限。 该参数值预期为通过 `chmod` 和 `umask` 系统调用接受的格式指定的数字模式。 （为了使用习惯八进制格式的数字必须以 `0` (zero)开始）。

默认的权限 `0600` ，只意味着 服务器拥有者可以读取或写入日志文件。其他较普遍 有用的设置是 `0640` ，让拥有者组成员来读取文件。 但是请注意，为了充分使用这种设置，你需要改变log_directory到集群数据目录之外的 某处存储这些文件。无论如何，使日志文件全局可读是不明智的，因为它们可能包含敏感数据。

这个参数可以在 `postgresql.conf` 文件或者服务器命令行上设置。

`log_rotation_age` (`integer`)

在打开了 `logging_collector` 的时候，这个选项设置一个独立日志文件的最大生存期。 在数值指定的分钟过去之后，将创建一个新的日志文件。设置为零可以关闭以时间为基础的新日志文件的创建。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行设置。

`log_rotation_size` (`integer`)

在打开了 `logging_collector` 的时候，这个选项设置一个独立的日志文件的最大尺寸。 在数值指定的千字节写入日志文件之后，将会创建一个新的日志文件。 设置为零可以关闭以尺寸为基础的新日志文件的创建。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行上设置。

`log_truncate_on_rotation` (`boolean`)

在打开了 `logging_collector` 的时候，这个选项将导致PostgreSQL 覆盖而不是附加到任何同名的现有日志文件上。 不过，覆盖只是发生在基于时间滚动而创建的新文件上，而不是在服务器启动的时候或者以尺寸为基础的滚动上。 如果为 off，将始终向已存在的文件结尾追加。 比如， 使用这个选项和类似 `postgresql-%H.log` 这样的 `log_filename` 设置将导致生成 24个按小时生成的日志文件然后在这些文件上循环。 这个选项只能在 `postgresql.conf` 文件里或者在服务器启动的时候设置。

例子：保留 7 天的日志，每天一个日志文件，叫做 `server_log.Mon`，`server_log.Tue` 等等，并且上周的日志会自动被这周的日志覆盖。把 `log_filename` 设置为 `server_log.%a`，把 `log_truncate_on_rotation` 设置为 `on`，并且把 `log_rotation_age` 设置为 `1440`。

例子：保留 24 小时的日志，每小时一个日志，但是如果日志文件尺寸大于 1GB 也旋转日志。 把 `log_filename` 设置为 `server_log.%H%M`，`log_truncate_on_rotation` 设置为 `on`，`log_rotation_age` 设置为 `60` 并且把 `log_rotation_size` 设置为 `1000000`。 在 `log_filename` 里包含 `%M` 允许任何尺寸驱动的旋转选取一个和开始的文件名同小时数但是名字不同的文件。

`syslog_facility` (`enum`)

如果向syslog进行记录，那么这个选项判断要使用的syslog"设施"。 你可以从 `LOCAL0`，`LOCAL1`，`LOCAL2`，`LOCAL3`，`LOCAL4`，`LOCAL5`，`LOCAL6`，`LOCAL7` 中选择。缺省是 `LOCAL0`。 又见你的系统的syslog守护进程文档。 这个选项只能在 `postgresql.conf` 文件里或者服务器启动的时候设置。

`syslog_ident` (`string`)

如果向syslog进行记录，这个选项决定用于在syslog日志中标识PostgreSQL的程序名。 缺省是 `postgres`。这个选项只能在 `postgresql.conf` 文件里或者服务器启动的时候设置。

`event_source` (`string`)

当启用记录event log时，此参数 确定用于识别PostgreSQL消息日志的程序名称。 缺省是 `PostgreSQL` 。 该参数只能在 `postgresql.conf` 文件或者服务器命令行上设置。

18.8.2. 什么时候记录日志

`client_min_messages` (`enum`)

这个选项控制哪些信息发送到客户端。 有效的数值是 `DEBUG5` , `DEBUG4` , `DEBUG3` , `DEBUG2` , `DEBUG1` , `LOG` , `NOTICE` , `WARNING` , `ERROR` , `FATAL` , 和 `PANIC` 。 每个级别包含所有它后面的级别， 级别越靠后， 发送的信息越少。缺省是 `NOTICE` 。 需要注意的是这里的 `LOG` 和 `log_min_messages` 里的级别不同。

`log_min_messages` (`enum`)

控制写到服务器日志里的信息的详细程度。 有效值是 `DEBUG5` , `DEBUG4` , `DEBUG3` , `DEBUG2` , `DEBUG1` , `INFO` , `NOTICE` , `WARNING` , `ERROR` , `LOG` , `FATAL` 和 `PANIC` 。 每个级别都包含它后面的级别。越靠后的数值发往服务器日志的信息越少。缺省是 `WARNING` 。 需要注意的是这里的 `LOG` 和 `client_min_messages` 里的级别不同。只有超级用户可以修改这个设置。

`log_min_error_statement` (`enum`)

控制在服务器日志里输出哪一条导致错误条件的SQL语句。 所有导致一个特定级别(或者更高级别)错误的 SQL 语句都要被记录。 有效的值有 `DEBUG5` , `DEBUG4` , `DEBUG3` , `DEBUG2` , `DEBUG1` , `INFO` , `NOTICE` , `WARNING` , `ERROR` , `LOG` , `FATAL` 和 `PANIC` 。 缺省是 `ERROR` ， 表示所有导致错误、日志信息，致命错误、恐慌的SQL语句都将被记录。 设置为 `PANIC` 表示把这个特性关闭。只有超级用户可以改变这个设置。

`log_min_duration_statement` (`integer`)

如果某个语句的持续时间大于或者等于这个毫秒数，那么在日志行上记录该语句及其持续时间。 设置为零将打印所有查询和他们的持续时间。设置为-1(缺省值)关闭这个功能。比如，如果你把它设置为 `250ms` ， 那么所有运行时间等于或者超过 250ms 的 SQL 语句都会被记录。 打开这个选项可以很方便地跟踪需要优化的查询。只有超级用户可以改变这个设置。

对于使用扩展查询协议的客户端，语法分析、邦定、执行每一步所花时间都分别记录。

> **Note:** 当此选项与log_statement同时使用时， 已经被 `log_statement` 记录的语句文本不会被重复记录。 如果没有使用syslog的话，推荐使用log_line_prefix记录 PID 或会话ID，这样就可以使用它们将语句消息连接耗时消息。

Table 18-1解释了PostgreSQL使用的 信息严重程度。如果日志输出发送到syslog或者Windows的 eventlog，则严重程度如下表所示。

Table 18-1. 信息严重级别

严重级别	用法	syslog	eventlog
DEBUG1..DEBUG5	提供开发人员使用的连续更多详细信息	DEBUG	INFORMATION
INFO	提供用户隐含要求的信息，比如在 VACUUM VERBOSE 过程输出的信息。	INFO	INFORMATION
NOTICE	提供可能对用户有帮助的信息，比如，长标识符的截断	NOTICE	INFORMATION
WARNING	提供可能问题的警告，比如在事务块范围之外的 COMMIT	NOTICE	WARNING
ERROR	报告导致当前命令退出的错误。	WARNING	ERROR
LOG	报告一些管理员感兴趣的信息，比如，检查点活跃性。	INFO	INFORMATION
FATAL	报告导致当前会话终止的错误。	ERR	ERROR
PANIC	报告导致所有数据库会话退出的错误。	CRIT	ERROR

18.8.3. 记录什么

`application_name` (`string`)

该 `application_name` 可以是小于 `NAMEDATALEN` 字符 （标准构建64个字符）的任何字符串。它通常是由应用程序连接到服务器设置。 该名称将显示在 `pg_stat_activity` 视图中 并包含在CSV格式的日志项中。它也可以被包括在 通过log_line_prefix参数的常规日志项中。 只有可打印的ASCII字符可能用在 `application_name` 值中。其它字符会 用问号(`?`)代替。

`debug_print_parse` (`boolean`) `debug_print_rewritten` (`boolean`) `debug_print_plan` (`boolean`)

这些选项打开各种调试输出。当设置，它们打印生成的解析树，查询重写输出， 或每个执行查询的执行计划。 这些消息在 `LOG` 消息级别发出的， 所以默认情况下它们会出现在服务器日志中，但不会被发送到客户端。 您可以通过调整client_min_messages和/或log_min_messages改变。这些参数缺省都是关闭的。

`debug_pretty_print` (`boolean`)

当设置时， `debug_pretty_print` 缩进 `debug_print_parse` ， `debug_print_rewritten` 或者 `debug_print_plan` 产生的消息， 这样更加可读，但是，当关闭它时，比"紧凑型"格式更长的输出，缺省是打开的。

`log_checkpoints` (`boolean`)

导致检查点和重启点被记录在服务器日志中。 一些统计都包含在日志信息中，包括缓冲区写入数量和编写它们的花费的时间。 这些参数只能在 `postgresql.conf` 文件或者服务器命令行上设置，缺省是off。

`log_connections` (`boolean`)

导致记录到服务器的每个尝试连接，以及成功完成客户端认证。 该参数在会话开始之后不能改变，缺省是off。

> **Note:** 某些客户端程序，如psql当确定是否需要密码的时候， 企图连接两次。所以复制"连接收到"的消息不一定表示有问题。

`log_disconnections` (`boolean`)

这个选项类似 `log_connections` ，但是在会话结束的时候在服务器日志里输出一行。 并且包含会话持续时间。 缺省是关闭的。这个参数在会话开始之后不能被改变。

`log_duration` (`boolean`)

记录每个已完成语句的持续时间。默认值是 `off` 。只有超级用户可以改变这个设置。

对于使用扩展查询协议的客户端，语法分析、邦定、执行每一步所花时间都分别记录。

> **Note:** 设置为0时该选项与log_min_duration_statement 的不同之处在于 `log_min_duration_statement` 强制记录查询文本。但是这个选项不可以。因此，如果 `log_duration` 为 `on` 并且 `log_min_duration_statement` 大于零将记录所有持续时间，但是仅记录那些超过阈值的语句。 这可以用于在高负载情况下搜集统计信息。

`log_error_verbosity` (`enum`)

控制记录的每条信息写到服务器日志里的详细程度。 有效的值是 `TERSE` , `DEFAULT` 和 `VERBOSE` ， 逐个向显示的信息里增加更多的字段。 `TERSE` 包含 `DETAIL` 的记录， `HINT` , `QUERY` 和 `CONTEXT` 错误信息。 `VERBOSE` 输出包含 `SQLSTATE` 错误代码(参见Appendix A)以及源代码文件名称，函数名称，以及产生错误的行数。 只有超级用户可以改变这个设置。

`log_hostname` (`boolean`)

缺省时，连接日志只记录所连接主机的IP地址。打开这个选项导致同时记录主机名。 请注意，这样有可能带来一些不可忽略的性能损失(取决于你的名字解析的设置)。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行设置。

`log_line_prefix` (`string`)

这是一个 `printf` 风格的字符串，在日志的每行开头输出。 `%` 字符开始"转义序列"被如下所述状态信息替换。 无法识别的转义被忽略。其它字符都直接拷贝到日志行中。 有些逃逸只被会话进程识别，被后端进程忽略，比如主服务器进程。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行设置。 缺省是空字符串。

逃逸	效果	仅用于会话
`%a`	Application name	yes
`%u`	User name	yes
`%d`	Database name	yes
`%r`	Remote host name or IP address, and remote port	yes
`%h`	Remote host name or IP address	yes
`%p`	Process ID	no
`%t`	Time stamp without milliseconds	no
`%m`	Time stamp with milliseconds	no
`%i`	Command tag: type of session's current command	yes
`%e`	SQLSTATE error code	no
`%c`	Session ID: see below	no
`%l`	Number of the log line for each session or process, starting at 1	no
`%s`	Process start time stamp	no
`%v`	Virtual transaction ID (backendID/localXID)	no
`%x`	Transaction ID (0 if none is assigned)	no
`%q`	Produces no output, but tells non-session processes to stop at this point in the string; ignored by session processes	no
`%%`	Literal `%`	no

`%c` 逃逸打印唯一会话标识符， 由两个点号分隔的4字节的十六进制数字（没有前导零）组成。 数字是该过程的开始时间和进程ID， 所以 `%c` 也可以用做一种打印这些项目的节约空间的方法。 例如，为了从 `pg_stat_activity` 中生成会话标识符，使用这个查询：

```
SELECT to_hex(EXTRACT(EPOCH FROM backend_start)::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

> **Tip:** 如果你设置 `log_line_prefix` 的非空值，你应该经常使其最后一个字符是一个空格，提供来自日志行的其余部分的视觉分离。一个标点符号也可以使用。
>
> **Tip:** Syslog产生自身时间戳和进程ID信息，如果你记录到syslog， 所以你可能不想包括那些逃脱。

`log_lock_waits` (`boolean`)

当会话等待比deadlock_timeout获得锁更长的时间时，控制是否产生一个日志消息。 决定如果锁等待造成很差的性能，这是很有用处的。默认是 `off` 。

`log_statement` (`enum`)

控制记录哪些SQL语句。有效的值是 `none` (off), `ddl` , `mod` 和 `all` (所有语句)。 `ddl` 记录所有数据定义命令，比如 `CREATE` , `ALTER` 和 `DROP` 语句。 `mod` 记录所有 `ddl` 语句， 加上数据修改语句 `INSERT` , `UPDATE` , `DELETE` , `TRUNCATE` , 和 `COPY FROM` 。如果所包含的命令类型吻合，那么 `PREPARE` , `EXECUTE` 和 `EXPLAIN ANALYZE` 语句也同样被记录。 对于使用扩展查询协议的客户端，记录发生在接受到扩展信息并包含邦定参数(内置单引号要双写)的时候。

缺省是 `none` 。只有超级用户可以改变这个设置。

> **Note:** 即使设置了 `log_statement` = `all` ， 包含简单语法错误的语句也不会被记录。 因为仅在完成基本的语法分析并确定了语句类型之后才记录日志。在使用扩展查询协议的情况下， 在执行阶段之前(语法分析或规划阶段)同样不会记录。 将 `log_min_error_statement` 设为 `ERROR` 或更低才能记录这些语句。

`log_temp_files` (`integer`)

控件的临时文件名称和大小。临时文件可以创建的分类，哈希，和临时的查询结果。 当它被删除时，一个日志项有利于每个临时文件。 零值记录所有临时文件的信息， 同时正值记录文件大小大于或等于千字节指定数量。 默认设置是-1，禁用这样的日志。只有超级用户可以更改此设置。

`log_timezone` (`string`)

设置用于写入服务器日志的时间戳的时区。不像TimeZone，这个值是簇范围， 因此，所有会话将持续报告时间戳。默认值 `GMT` ， 但这通常被 `postgresql.conf` 覆盖；initdb 将安装与其系统环境一致的设置。参见Section 8.5.3获取更多信息。 这个参数只能在 `postgresql.conf` 文件或者服务器命令行上进行。

18.8.4. 使用CSV-格式日志输出

在 `log_destination` 列表中包含 `csvlog` 提供了一种便捷的方式导入日志文件到一个数据库表。 此选项在逗号分隔值(CSV)中发出日志行，这些列为： time stamp with milliseconds, user name, database name, process ID, client host:port number, session ID, per-session line number, command tag, session start time, virtual transaction ID, regular transaction ID, error severity, SQLSTATE code, error message, error message detail, hint, 导致错误的内部查询（如果有），字符计算错误位置， 包含错误文本，导致错误的用户查询（如果有并且通过 `log_min_error_statement` 启动）， 字符计算错误位置。其中有PostgreSQL源代码错误的位置（如果 `log_error_verbosity` 设置为 `verbose` ）， 以及应用程序的名称。这是一个用于存储CSV格式的日志输出样本表定义：

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

使用 `COPY FROM` 命令，将日志文件导入到这个表中：

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

你需要做几件事情简化导入CSV日志文件：

1. 设置 `log_filename` 和 `log_rotation_age` 为你的日志文件 提供一个一致的，可预见的命名方式。 这让你预测哪些文件的名称以及独立日志文件是完整的，因此可以准备导入。
2. 设置 `log_rotation_size` 为0以禁用基于大小的日志旋转， 因为它很难预测日志文件名。
3. 设置 `log_truncate_on_rotation` 到 `on` ，以致于 旧的日志数据不与同一文件的新数据混合。
4. 上面表定义包括主密钥规范。这是为了防止意外导入相同的有用信息两次。 `COPY` 命令要求所有的数据一次导入，所以任何错误都将导致整个导入过程失败。 如果你导入部分日志文件，当它完成时，然后再导入该文件，主键冲突会导致导入过程失败。 等待直到日志是完整的，并且导入之前关闭。 该程序也将防止意外导入尚未完全写入的局部行，这也将导致 `COPY` 失败。

18.9. 运行时统计

18.9.1. 查询和索引统计收集器

下面的参数控制服务器范围的统计搜集特性。如果启用了统计搜集， 那么生成的数据可以通过 `pg_stat` 和 `pg_statio` 系统视图家族访问。 参见Chapter 27获取更多信息。

`track_activities` (`boolean`)

统计每个会话执行的命令及其开始执行的时间。这个选项缺省是开启的。 请注意即使把它打开，这个信息也不是所有用户都可见的， 只有超级用户和会话的所有者才能看到；因此它不应该是个安全漏洞。 只有超级用户可以改变这个设置。

`track_activity_query_size` (`integer`)

指定跟踪当前执行命令的预留字节数， 为了每个活动会话，以及 `pg_stat_activity` . `query` 字段。 默认值是1024。这个参数只能在服务器启动时设置。

`track_counts` (`boolean`)

打开数据库活动的统计收集。此参数缺省是开启的，因为自动清理守护进程需要收集信息。只有超级用户可以更改此设置。

`track_io_timing` (`boolean`)

启动数据库定时I/O调用。此参数缺省是关闭的，因为它会反复查询操作系统当前的时间， 这可能会导致某些平台的显著开销。 您可以使用pg_test_timing工具测量系统上的定时开销。在EXPLAIN的输出中通过， pg_stat_statements使用 `BUFFERS` 选项时。 I/O时序信息显示在pg_stat_database上， 只有超级用户才能更改此设置。

`track_functions` (`enum`)

启用函数调用计数和时间跟踪。指定 `pl` 仅跟踪过程语言函数， `all` 跟踪SQL和C语言函数。 默认是 `none` ，禁用函数统计跟踪。 只有超级用户可以更改此设置。

> **Note:** 足够简单的以便"内联"到调用查询的SQL语言函数将不被跟踪， 而不管这些设置。

`update_process_title` (`boolean`)

服务器每收到一个新的SQL命令就更新进程标题。 进程标题可以通过 `ps` 命令或Windows下的进程管理器查看。 只有超级用户可以改变这个设置。

`stats_temp_directory` (`string`)

设置存储临时统计数据的目录。这可以是相对于数据目录的相对路径或绝对路径。 缺省是 `pg_stat_tmp` 。 指向基于RAM文件系统将减少物理I/O要求，并可能导致性能提升。 此参数只能在 `postgresql.conf` 文件或者服务器命令行上设置。

18.9.2. 统计监控

`log_statement_stats` (`boolean`) `log_parser_stats` (`boolean`) `log_planner_stats` (`boolean`) `log_executor_stats` (`boolean`)

对每条查询，向服务器日志里输出相应模块的性能统计。 这是原始的剖析工具。类似于Unix `getrusage()` 操作系统工具。 `log_statement_stats` 报告总的语言统计， 而其它的报告针对每个模块的统计。 `log_statement_stats` 不能和其它任何针对每个模块统计的选项一起打开。所有这些选项都是缺省关闭的。 只有超级用户才能修改这些设置。

18.10. 自动清理

这些设置控制自动清理的缺省行为。请参阅Section 23.1.6获取更多信息。

`autovacuum` (`boolean`)

控制服务器是否应该启动autovacuum守护进程。缺省是关闭的。然而， track_counts还必须启用自动清理工作。 这个选项只能在 `postgresql.conf` 文件里或者是服务器命令行中设置。

请注意，即使禁用该参数，如果有必要避免事务ID重叠，系统仍将启动自动清理进程。 请参阅Section 23.1.5获取更多详细信息。

`log_autovacuum_min_duration` (`integer`)

如果它运行至少毫秒指定数，导致记录自动清理所执行的每个动作。 此设置为零记录所有自动清理操作。减一（默认）禁用日志记录 自动清理动作。例如，如果您将其设置为 `250ms` ，那么将记录运行250ms或更长时间的所有自动清理和分析。 此外，当这个参数设置为除 `-1` 外的其他任何值， 如果由于冲突锁的存在而忽略自动清理操作，则记录这条消息。 启用此参数可以有益于跟踪自动清理活动。这个设置只能在 `postgresql.conf` 文件或者服务器命令行上设置。

`autovacuum_max_workers` (`integer`)

指定自动清理进程的最大数（除了自动清理发射器）， 它可以在任一时刻运行。 默认是三。 这个参数只能在服务器启动时设置。

`autovacuum_naptime` (`integer`)

声明运行在任何给定数据库上的autovacuum之间的最小延迟。在每次运行的周期里， 守护进程都会检查一个数据库， 并根据需要为该数据库的表发出 `VACUUM` 和 `ANALYZE` 命令。 这个延迟是以秒计的，缺省为1分钟(`1min`)。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。

`autovacuum_vacuum_threshold` (`integer`)

声明在任何表里触发 `VACUUM` 所需最小的行更新或删除数量。缺省是500。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。 此处的设置可以被改变存储参数的独立的表覆盖。

`autovacuum_analyze_threshold` (`integer`)

声明在任何表里触发 `ANALYZE` 所需最小的行插入、更新、删除数量。缺省是50。 这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。 此处的设置可以被改变存储参数的独立表覆盖。

`autovacuum_vacuum_scale_factor` (`floating point`)

声明在判断是否触发一个 `VACUUM` 时增加到 `autovacuum_vacuum_threshold` 参数里面的表尺寸的分数。 缺省是0.2(表大小的20%)。这个选项只能在 `postgresql.conf` 文件里或者服务器命令行中设置。 此处的设置可以被改变存储参数的独立表覆盖。

`autovacuum_analyze_scale_factor` (`floating point`)

声明在判断是否触发一个 `ANALYZE` 时增加到 `autovacuum_analyze_threshold` 参数里面的表尺寸的分数。缺省是0.1(表大小的10%)。 这个选项只能在 `postgresql.conf` 文件里或者服务器启动的时候设置。 此处的设置可以被改变存储参数的独立表覆盖。

`autovacuum_freeze_max_age` (`integer`)

指定表的 `pg_class` 在事务中的最大寿命。 `relfrozenxid` 字段能够在强制 `VACUUM` 操作以防止事务 ID在表内循环重复之前完成。需要注意的是，即使autovacuum被禁止系统也会调用autovacuum 进程来防止循环重复。

自动清理允许删除来自 `pg_clog` 子目录的旧文件， 默认值是相对低于200百万事务。该参数只能在服务器启动时设置， 但是此处的设置可以通过改变存储参数独立表减少。 更多信息请参见Section 23.1.5。

`autovacuum_vacuum_cost_delay` (`integer`)

声明将在自动 `VACUUM` 操作里使用的开销延迟数值。 声明-1将使用普通的vacuum_cost_delay数值。 缺省值是20毫秒。 这个选项只能在 `postgresql.conf` 文件里或者在服务器启动的时候设置。 此处的设置可以被改变存储参数的独立表覆盖。

`autovacuum_vacuum_cost_limit` (`integer`)

声明将在自动 `VACUUM` 操作里使用的开销限制数值。 -1(缺省值)将使用普通的vacuum_cost_limit数值。 要注意的是值将按比例分配给运行autovacuum工作者， 如果有一个以上，这样每个工作者限制总和不会超过此变量的极限。 这个选项只能在 `postgresql.conf` 文件里或者在服务器启动的时候设置。 此处的设置可以通过改变存储参数独立表覆盖。

18.11. 客户端连接缺省

18.11.1. 语句行为

`search_path` (`string`)

这个变量声明模式的搜索顺序，在一个被引用对象(表、数据类型、函数等)只是一个简单名字， 没有声明模式时需要这样的搜索。如果在另外一个模式里有一个相同的对象名， 那么使用在这个搜索路径中找到的第一个。 一个不在搜索路径中任何一个模式里出现的对象只能通过其所在模式的全称(打点的)名字来声明。

`search_path` 的值必需是一个逗号分隔的模式名列表。 它不是一个现有的模式名，或者用户不具有 `USAGE` 权限的模式，都将被自动忽略。

如果列表项之一是特殊名称 `$user` ， 那么通过 `SESSION_USER` 返回的名称模式被取代， 如果有这样一个模式并且用户已经有 `USAGE` 权限。 （如果不是，那么忽略 `$user` 。）

系统表模式 `pg_catalog` 总是被搜索，不管是否在搜索路径。如果在路径中， 那么按照路径指定的顺序搜索， 如果 `pg_catalog` 不在路径中，那么将在任何路径之前搜索。

同样，如果它存在，那么当前会话的临时表模式， `pg_temp_``_nnn_` 总是被搜索。 它可以通过使用别名 `pg_temp` 被明确地列在该路径中。如果没有列在路径中，那么它首先被搜索（甚至 `pg_catalog` 之前）。 但是，临时模式只搜索关系（表，视图，序列等）以及数据类型名称。 它从来没有搜索函数或运算符名称。

如果创建对象时没有声明特定的目标模式，那么它将被放进 `search_path` 中的第一个模式。 如果搜索路径是空的，那么会报告一个错误。

这个参数的缺省值是 `"$user", public` 。 这样就支持共享使用一个数据库(没有用户拥有私有模式，所有人都共享使用 `public`)、 私有的针对每个用户的模式、以及两者的组合。 其它效果可以通过全局或者针对每个用户修改搜索路径设置获取。

搜索路径当前值可以用SQL函数 `current_schemas` （参阅Section 9.25） 检查。它和检查 `search_path` 的值不太一样，因为 `current_schemas` 显示的是在 `search_path` 里出现的项如何被处理。

有关模式处理的更多信息，参阅Section 5.7。

`default_tablespace` (`string`)

这个变量声明当 `CREATE` 命令没有明确声明表空间时， 所创建对象(表和索引等)的缺省表空间。

值要么是一个表空间的名字，要么是一个表明使用当前数据库缺省表空间的空字符串。 如果这个数值和任意现存表空间的名字都不匹配， 那么PostgreSQL将自动使用当前数据库的缺省表空间。 如果声明非缺省表空间，用户必须有 `CREATE` 权限，或者创建尝试将失败。

这个变量不用于临时表;对他们来说， temp_tablespaces提供咨询。

这个变量在创建数据库时也没有使用。默认情况下，新的数据库继承了从 模板数据库复制的表空间设置。

关于表空间的更多信息，请参阅Section 21.6。

`temp_tablespaces` (`string`)

当 `CREATE` 命令不明确指定一个表空间时， 这个变量指定要在其中创建临时对象的表空间（临时表和临时表的索引）。 出于此目的的临时文件，比如排序大型数据集，也在这些表空间中创建。

该值是表空间名称的列表。当列表中有一个以上名称时， PostgreSQL每次临时对象被创建时选择一个列表中的随机数； 除了在一个事务中之外，先后创建临时 对象放置在列表连续的表空间中。 如果列表中选定的元素是一个空字符串， PostgreSQL会自动使用当前数据库的缺省表空间。

当 `temp_tablespaces` 交互设置时，指定 不存在的表空间是一个错误，为 不具有 `CREATE` 权限的用户声明一个表空间。但是， 当使用事先设定的值时，则忽略不存在的表空间， 因为对于缺少 `CREATE` 权限的用户是一个表空间。 特别是，该规则适用于使用 `postgresql.conf` 设置的值时。

默认值是一个空字符串，这会导致所有临时对象在当前数据库缺省表空间被创建。

参阅default_tablespace。

`check_function_bodies` (`boolean`)

这个参数通常是on。设置为 `off` 表示在CREATE FUNCTION之间关闭函数体字符串的合法性检查。 关闭合法性检查有时候会有用，比如避免从转储中恢复函数定义时向前引用的问题。

`default_transaction_isolation` (`enum`)

每个SQL事务都有一个隔离级别，可以是"读未提交"，"读已提交"， "可重复读"或者是"可串性化"。 这个参数控制每个新事务的缺省隔离级别。缺省是"读已提交"。

参考Chapter 13和SET TRANSACTION获取更多信息。

`default_transaction_read_only` (`boolean`)

只读的SQL事务不能修改非临时表。这个参数控制每个新事务的只读状态。缺省是 `off` (读/写)。

参考SET TRANSACTION获取更多信息。

`default_transaction_deferrable` (`boolean`)

当在 `可串行化` 隔离级别下运行时， 延迟的只读SQL事务允许继续进行之前可能会延迟。 然而，一旦开始执行不会产生任何开销要求，以确保串行化; 所以串行化代码没有理由迫使它终止，因为并发更新， 这使这些选项适合长时间运行只读事务。

此参数控制每个新事务缺省延迟状态。 它目前对读写事务或那些比 `可串行化` 低的隔离级别下的操作没有影响。 缺省是 `off` 。

参阅SET TRANSACTION获取更多详细信息。

`session_replication_role` (`enum`)

控制当前会话复制相关的触发器和规则。设置此变量需要 超级用户权限，并导致丢弃任何以前缓存查询规划。 可能的值是 `origin` (缺省)， `replica` 和 `local` 。 参阅ALTER TABLE获取更多信息。

`statement_timeout` (`integer`)

退出任何使用了超过此参数指定时间(毫秒)的语句，从服务器收到命令时开始计时。 如果 `log_min_error_statement` 设置为 `ERROR` 或者更低， 那么也会在日志中记录超时。 零值(缺省)关闭这个计时器。

不推荐设置 `postgresql.conf` 中的 `statement_timeout` ， 因为它影响所有的会话。

`lock_timeout` (`integer`)

当试图获取表、索引、行或其他数据库对象的锁时，终止等待时间超过指定毫秒数的任何语句。 时间限制分别适用于每个锁获取尝试。该限制适用于明确 锁定请求（如 `LOCK TABLE` 或者没有 `NOWAIT` 的 `SELECT FOR UPDATE` ） 以及隐式获取的锁。 如果 `log_min_error_statement` 设置为 `ERROR` 或更低， 则记录超时的语句。零值（缺省值）关闭它。

不像 `statement_timeout` ，此超时只能发生在等待锁的时候。 请注意，如果 `statement_timeout` 是非零， 设置 `lock_timeout` 相同或更大的值是相当没有意义的， 因为该语句超时总是会首先触发。

不推荐设置 `postgresql.conf` 中的 `lock_timeout` ， 因为它影响所有的会话。

`vacuum_freeze_table_age` (`integer`)

如果该表 `pg_class` . `relfrozenxid` 字段已达到此设置中指定的时间， `VACUUM` 执行全表扫描。 默认值是1.5亿个事务。虽然用户可以从零到十亿设置此值。 `VACUUM` 会默默的限制autovacuum_freeze_max_age的95%的有效值， 从而使定期手动的 `VACUUM` 在自动清理该表之前有运行机会。 详细信息请见Section 23.1.5。

`vacuum_freeze_min_age` (`integer`)

指定 `VACUUM` 在扫描一个表时用于判断是否用 `FrozenXID` 替换事务ID的中断寿命(在同一个事务中)。 缺省值为50百万。虽然用户可以指定一个 0-1000000000之间的值， 但是 `VACUUM` 将会悄无声息的将有效值限制在autovacuum_freeze_max_age的一半之内。 更多信息参见Section 23.1.5。

`bytea_output` (`enum`)

设置 `bytea` 类型值的输出格式。 有效值为 `hex` (缺省) 和 `escape` (传统PostgreSQL 格式)。参见Section 8.4获取更多信息。 不管这些设置，其中 `bytea` 类型总是接受这两种格式的输入。

`xmlbinary` (`enum`)

设置二进制值是如何在XML中进行编码的。 这适用于当 `bytea` 值通过 `xmlelement` 或者 `xmlforest` 函数被转换为XML。 可能的值是 `base64` 和 `hex` ， 这在XML模式标准中定义。 默认值是 `base64` 。 关于XML相关的函数的进一步信息，请参阅Section 9.14。

这里的实际选择主要是口味问题， 只有在客户端应用程序中可能存在某些限制约束。 这两种方法都支持所有可能的值， 虽然十六进制编码将会比base64编码大一些。

`xmloption` (`enum`)

当XML和字符串值之间进行转换时，设置 `DOCUMENT` 或 `CONTENT` 是否是隐含的。 参见Section 8.13获取更多详细信息。有效 值 `DOCUMENT` 和 `CONTENT` 。缺省是 `CONTENT` 。

按照SQL标准，设置这个选项的命令是：

```
SET XML OPTION { DOCUMENT | CONTENT };
```

这种语法在PostgreSQL中是可用的。

18.11.2. 区域和格式化

`DateStyle` (`string`)

设置日期和时间值的显示格式，以及有歧义的输入值的解析规则。 由于历史原因，这个变量包含两个独立的部分：输出格式声明(`ISO` ， `Postgres` ， `SQL` 或者 `German`)、 输入输出的年/月/日顺序(`DMY` ， `MDY` 或者 `YMD`)。 这两个可以独立设置或者一起设置。 关键字 `Euro` 和 `European` 等价于 `DMY` ； 关键字 `US` ， `NonEuro` 和 `NonEuropean` 等价于 `MDY` 。 参阅Section 8.5获取更多信息。内置缺省是 `ISO, MDY` ， 但是initdb将在初始化配置文件时根据 `lc_time` 选择一个合适的默认设置。

`IntervalStyle` (`enum`)

设置区间值的显示格式。 `sql_standard` 将产生输出匹配SQL标准时间间隔。 `postgres` 的值(默认值)会产生输出匹配PostgreSQL8.4之前的版本。 当DateStyle参数设置为 `ISO` 时。 `postgres_verbose` 将产生输出匹配PostgreSQL8.4之前的版本。 当 `DateStyle` 参数被设置成非- `ISO` 输出时。 `iso_8601` 将产生输出匹配4.4.3.2节中ISO 8601定义的时间间隔格式。

`IntervalStyle` 参数也会影响不明确的间隔输入的说明。 参阅Section 8.5.4获取更多详细信息。

`TimeZone` (`string`)

设置用于显示和解析时间戳的时区。内置缺省值 `GMT` ， 但是，这通常被 `postgresql.conf` 改写；initdb 将安装设置对应它的系统环境。 意味着使用系统环境声明的时区。 参阅Section 8.5.3获取更多信息。

`timezone_abbreviations` (`string`)

设置服务器接受日期时间输入中使用的时区缩写集合。缺省值 `'Default'` ， 在全世界大多数地方都能工作的很好。 另外的可用值还有 `'Australia'` 和 `'India'` 等其它值。 参见Appendix B以获取更多信息。

`extra_float_digits` (`integer`)

这个参数为浮点数值调整显示的数据位数，浮点类型包括 `float4` ， `float8` 以及几何数据类型。 参数值加在标准的数据位数上(`FLT_DIG` 或者 `DBL_DIG` 中合适的)。 数值可以设置为最高3， 以包括部分关键的数据位；这个功能对转储那些需要精确恢复的浮点数据特别有用。 或者你也可以把它设置位负数以消除不需要的数据位。 参阅Section 8.1.3。

`client_encoding` (`string`)

设置客户端编码(字符集)。缺省使用数据库编码。 字符集通过Section 22.3.1里描述的PostgreSQL服务器支持。

`lc_messages` (`string`)

设置信息显示的语言。可接受的值是系统相关的； 参阅Section 22.1获取更多信息。如果这个变量设置为空字符串(缺省值)， 那么其值将以一种系统相关的方式从服务器的执行环境中继承。

在一些系统上，这个区域范畴并不存在，不过仍然允许设置这个变量， 只是不会有任何效果。同样，也有可能是所期望的语言的翻译信息不存在。 在这种情况下，你仍然能看到英文信息。

只有超级用户可以改变这个设置。因为它同时影响发送到服务器日志和客户端的信息。 并且不正确的值可能会掩盖服务器日志的可读性。

`lc_monetary` (`string`)

为格式化金额数量设置区域。比如用于 `to_char` 函数族。可接受的值是系统相关的； 参阅Section 22.1获取更多信息。如果这个变量设置为空字符串(缺省值)， 那么其值将以一种系统相关的方式从服务器的执行环境中继承。

`lc_numeric` (`string`)

设置用于格式化数字的区域，比如用于 `to_char` 函数族。可接受的值是系统相关的； 参阅Section 22.1获取更多信息。如果这个变量设置为空字符串(缺省值)， 那么其值将以一种系统相关的方式从服务器的执行环境中继承。

`lc_time` (`string`)

设置用于格式化日期和时间值的区域。比如 `to_char` 函数族， 可接受的值是系统相关的；参阅Section 22.1获取更多信息。 如果这个变量设置为空字符串(缺省值)， 那么其值将以一种系统相关的方式从服务器的执行环境中继承。

`default_text_search_config` (`string`)

选择文本搜索函数使用的那些变量的文本搜索配置没有声明配置的明确参数。 参见Chapter 12获取进一步信息。 内置缺省值是 `pg_catalog.simple` ， 如果配置匹配可以被识别的区域，则initdb将初始化该设置的配置文件， 它对应于已选择的 `lc_ctype` 区域。

18.11.3. 其他缺省

`dynamic_library_path` (`string`)

如果需要打开一个可以动态装载的模块并且在 `CREATE FUNCTION` 或者 `LOAD` 命令里面声明的名字没有目录部分(也就是说名字里不包含斜杠)， 那么系统将搜索这个目录以查找声明的文件。

用于 `dynamic_library_path` 的数值必须是一个冒号分隔 (或者是在Windows上分号分隔)的绝对路径列表。 如果一个路径名字以特殊变量 `$libdir` (PostgreSQL编译好的库目录)开头， 那么就替换为PostgreSQL发布提供的模块安装路径。 (使用 `pg_config --pkglibdir` 打印这个目录名)。比如：

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

或者是在Windows环境里：

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

这个参数的缺省值是 `'$libdir'` 。 如果把这个值设置为一个空字符串，则关闭自动路径搜索。

这个参数可以在运行时由超级用户修改，但是这么修改的设置只能保持到这个客户端连接的结束， 因此这个方法应该保留给开发用途使用。我们建议在 `postgresql.conf` 配置文件里设置。

`gin_fuzzy_search_limit` (`integer`)

GIN索引扫描返回的集合尺寸软上限。更多信息参见Section 57.4。

`local_preload_libraries` (`string`)

指定在开始连接前预先加载一个或多个共享库，多个库之间用逗号分隔。 所有的库名被转换为小写字母，除非双引号引用。 该参数不能在会话开始之后更改。

因为并非只有超级用户才能更改此选项， 因此只能加载安装的标准库目录下 `plugins` 子目录中的库文件， 数据库管理员有责任确保该目录中的库都是"安全的"。 `local_preload_libraries` 中指定的项可以明确含有该目录， 例如 `$libdir/plugins/mylib` ；也可以仅指定库的名字— 例如 `mylib` (等价于 `$libdir/plugins/mylib`)。

与shared_preload_libraries不同的是， 在会话开始时加载库比第一次使用时相比并不具有性能优势。 相反，这个特性的目的是为了调试或者测量在特定会话中不明确使用 `LOAD` 加载库时的性能。 例如针对某个用户将该参数设为 `ALTER ROLE SET` 来进行调试。

如果指定的库未找到，那么连接将失败。

每一个支持PostgreSQL的库都有一个"magic block"用于确保兼容性。 因此不支持PostgreSQL的库不能通过这个方法加载。

18.12. 锁管理

`deadlock_timeout` (`integer`)

以毫秒计的时间，用于设置在检查是否存在死锁条件之前等待的时间。 检查是否存在死锁条件是一个昂贵的过程，因此服务器不会在每次等待锁的时候都运行这个过程。 我们乐观地假设在生产应用中的死锁是不常出现的， 因此我们在开始询问是否可以解锁之前只等待一个相对较短的时间。 增加这个值就减少了浪费在无用的死锁检查上的时间，但是减慢了报告真正死锁错误的速度。 缺省是1秒(`1s`)，这可能是你能够耐心等待的最短时间。在一个重负载的服务器上， 你可能需要增大它。这个值的典型设置应该超过你的事务持续时间， 这样就可以减少在锁释放之前就开始死锁检查的问题。 只有超级用户可以改变这个设置。

当log_lock_waits被设置时， 此参数也决定在发出有关锁等待的日志信息之前的等待时间长度。 如果您正在尝试调查锁定延迟， 你可能想设置一个小于正常 `deadlock_timeout` 的值。

`max_locks_per_transaction` (`integer`)

共享的锁表的大小是以假设任意时刻最多只有 `max_locks_per_transaction` * (max_connections + max_prepared_transactions) 个独立的对象需要被锁住为基础进行计算的。 这个参数控制分配给每个事务的锁定对象平均数。 单独事务只要所有事务锁适合在锁表中都可以锁定多个对象。 这不是锁定行的数目， 该值是无限的。缺省值64，已经经历史证明是足够的了， 不过如果你有在一个事务里接触很多不同的表的查询，那么你就可能需要提高这个数值。 比如带有很多孩子的父表查询。 这个值只能在服务器启动的时候设置。

当运行备用服务器时，你必须将此参数设置为比主服务器上相同或更高的值。否则， 不允许在备用服务器进行查询。

`max_pred_locks_per_transaction` (`integer`)

共享谓词锁表跟踪锁定在 `max_pred_locks_per_transaction` * (max_connections + max_prepared_transactions)对象上（例如，表）; 因此，只是许多不同的对象更可以在任何一个时间锁定。 此参数控制对象锁定分配给每个事务的平均数; 个别事务可以锁定多个对象，只要所有事务的锁适合在锁表中。这不是 可以锁定的行数;该值是无限的。 在默认情况下，64对测试已经足够了， 如果在可串行化事务中你有接触许多不同表的客户，那么您可能需要增大这个值。 此参数只能在服务器启动时设置。

18.13. 版本和平台兼容性

18.13.1. 以前的PostgreSQL版本

`array_nulls` (`boolean`)

控制数组输入解析器是否将未用引号界定的 `NULL` 作为数组的一个NULL元素。 默认为 `on` 表示允许向数组中输入NULL值。但8.2之前的PostgreSQL版本不支持这么做， 因此将把 `NULL` 当作字符串"NULL"。如果希望向后兼容这种旧式行为， 那么可以设为 `off` 。

即使该值被设为 `off` 也仍然能够创建包含NULL值的数组。

`backslash_quote` (`enum`)

控制字符串文本中的单引号是否能够用 `\'` 来表示。 首选的符合SQL标准的方法是将其双写(`''`)，但是PostgreSQL 在历史上也可以用 `\'` 来表示。 不过使用 `\'` 容易导致安全漏洞，因为在某些多字节字符集中存在最后一个字节等于 `\` 的 ASCII 值的字符。 如果客户端代码没有做到正确逃逸，那么将会导致SQL注入攻击。 如果服务器拒绝使用反斜杠逃逸来表示单引号的查询，那么就可以避免这种风险。 `backslash_quote` 的可用值是 `on` (总是允许 `\'`), `off` (总是拒绝), `safe_encoding` (仅在客户端字符集编码不会在多字节字符末尾包含 `\` 的ASCII值时允许)。 `safe_encoding` 是缺省设置。

需要注意的是，在字符串文本符合SQL标准的情况下， `\` 没有任何其它含义。 这个参数影响的只是如何处理不符合标准的字符串文本， 包括明确的字符串逃逸语法(`E'...'`)。

`default_with_oids` (`boolean`)

这个选项控制 `CREATE TABLE` 和 `CREATE TABLE AS` 在既没有声明 `WITH OIDS` 也没有声明 `WITHOUT OIDS` 的情况下， 是否在新创建的表中包含OID字段。它还决定 `SELECT INTO` 创建的表里面是否包含OID。 参数缺省是 `off` ；在PostgreSQL 8.0之前缺省为on。

我们反对在用户表中使用 OID ，因此大多数安装应该关闭这个变量。 需要OID的表应该在创建表的时候声明 `WITH OIDS` 。 启用这个变量可以与不遵循这一行为的旧的应用程序兼容。

`escape_string_warning` (`boolean`)

打开的时候，如果在普通的字符串文本里(`'...'` 语法)出现了一个反斜扛(`\`)并且 `standard_conforming_strings` 被关闭，那么就会发出一个警告。缺省是 `on` 。

想要使用反斜杠作为逃逸的应用程序 应该使用逃逸字符串语法(`E'...'`)进行修改， 因为通的字符串缺省行为作为普通字符的反斜扛对待。 启动这个变量帮助找到需要修改的代码。

`lo_compat_privileges` (`boolean`)

在PostgreSQL9.0之前，大对象没有访问权限，因此，总是被所有用户可读可写。 设置这个变量到 `on` 禁用新权限检查，为了兼容先前版本。 缺省是 `off` 。只要超级用户可以改变此设置。

设置这些变量不会禁用所有与大对象相关的安全检查—仅仅是在PostgreSQL 9.0 已经改变了的缺省操作。比如， `lo_import()` 和 `lo_export()` 不管这些设置都需要超级用户权限。

`quote_all_identifiers` (`boolean`)

当数据库生成SQL，强制引用所有标示符，即使它们（当前）不是关键字。这也将影响 `EXPLAIN` 的输出以及像 `pg_get_viewdef` 函数的结果。 参见pg_dump和pg_dumpall 的 `--quote-all-identifiers` 选项。

`sql_inheritance` (`boolean`)

这个设置控制着未修饰的表引用是否包含继承的子表。缺省是 `on` ， 这意味着包含子表（因此，缺省假定 `*` 后缀）。如果返回 `off` ， 则不包含子表（因此，假定 `ONLY` 前缀）。SQL标准需要包含子表。 因此 `off` 设置不符合规范，但是它提供了PostgreSQL 7.1版本之前的兼容性。 参见Section 5.8获取更多详细信息。

不赞成关闭 `sql_inheritance` ，因为发现操作有错误而且违反SQL标准。 关于继承操作的讨论在手册中通常假定它是 `on` 。

`standard_conforming_strings` (`boolean`)

控制普通字符串文本(`'...'`)中是否按照SQL标准把反斜扛当普通文本。 PostgreSQL 9.1开始，缺省是 `on` （之前缺省是 `off` ）。 应用可以检查这个参数来判断字符串文本如何被处理。 这个参数的出现也建议明确使用逃逸字符串语法(`E'...'`)来逃逸字符。 如果应用希望反斜杠作为逃逸字符对待，则使用逃逸字符串语法(Section 4.1.2.2)。

`synchronize_seqscans` (`boolean`)

这允许大表进行顺序扫描以同步其他的，所以并发扫描读取同一时间的同一批， 从而共享I/O负载。 启用此功能后，扫描可能会在表中间开始， 然后"包装开始环绕"以覆盖所有的行， 从而同步已在进行的扫描活动。 这可能导致没有 `ORDER BY` 子句的查询返回排序行的不可预测变化。 设置这个参数为 `off` 确保预8.3行为， 其中一个顺序扫描总是从表的开头开始。缺省是 `on` 。

18.13.2. 平台和客户端兼容

`transform_null_equals` (`boolean`)

如果打开，那么表达式 `_expr_` = NULL 或者 `NULL =` `_expr_` 将被当做 `_expr_` IS NULL处理， 也就是说，如果 `_expr_` 得出 NULL 值则返回真，否则返回假。 正确的 SQL 标准兼容的 `_expr_` = NULL行为总是返回NULL(未知)。 因此这个选项缺省是 `off` 。

不过，在Microsoft Access里的过滤表单生成的查询好像使用的是 `_expr_` = NULL测试 NULL，因此， 如果你使用这个界面访问数据库，你可能想把这个选项打开。 因为形如 `_expr_` = NULL的表达式总是返回NULL（使用SQL标准说明）， 它们不是很有用而且在应用中也不常见， 因此这个选项实际上没有什么害处。但是新用户常常在涉及NULL的表达式语义上感到糊涂， 因此缺省时不打开这个选项。

请注意这个选项只影响 `= NULL` 形式， 不包括其它比较操作符或者其它与一些涉及等号操作符的表达式计算(比如 `IN`)。 因此，这个选项不是垃圾程序的普遍修复。

请参考Section 9.2获取相关信息。

18.14. Error Handling

`exit_on_error` (`boolean`)

如果为真，那么任何错误都将终止当前事务。缺省时，设置为假，所以只有致命错误将终止会话。

`restart_after_crash` (`boolean`)

当缺省设置为真时，PostgreSQL将在后端崩溃后自动重新初始化。 设置这些值为真往往是优化数据库的最佳方式。然而，在某些情况下，比如当PostgreSQL 正由集群调用时，它可能禁用启动非常有益，使得集群可以得到控制并采取其认为适当的任何行动。

18.15. 预置选项

下面的"参数"是只读的，它们是在编译或安装PostgreSQL的时候决定的。 因此，他们被排除在了 `postgresql.conf` 文件之外。 这些选项报告各种PostgreSQL某些应用可能感兴趣的行为， 特别是管理性的前端。

`block_size` (`integer`)

报告磁盘块的大小。它是由编译服务器时 `BLCKSZ` 的值确定的。缺省值是 8192 字节。 有些配置变量的含义(比如shared_buffers)会被 `block_size` 影响。 参阅Section 18.4获取信息。

`integer_datetimes` (`boolean`)

报告PostgreSQL 是否在编译时打开了 64 位整数日期和时间。 这是当编译PostgreSQL时，通过配置选项 `--disable-integer-datetimes` 禁用的。 缺省值是 `on` 。

`lc_collate` (`string`)

报告文本数据排序使用的区域。参阅Section 22.1获取更多信息。 该值是在初始化数据库集群的时候确定的。

`lc_ctype` (`string`)

报告决定字符分类的区域。参阅Section 22.1获取更多信息。 该值是在数据库集群初始化的时候决定的。 通常它和 `lc_collate` 一样，但是可以为特殊应用设置成不同的值。

`max_function_args` (`integer`)

报告函数参数的最大个数。它是由编译服务器时的 `FUNC_MAX_ARGS` 值决定的。缺省是100。

`max_identifier_length` (`integer`)

报告最大标识符长度。它是由编译服务器时的 `NAMEDATALEN` 值减一决定的。 `NAMEDATALEN` 的缺省值是 64 ； 因此 `max_identifier_length` 的缺省是63。 当使用多字节编码时小于63字符。

`max_index_keys` (`integer`)

报告最大索引键字的个数。它是由编译服务器时的 `INDEX_MAX_KEYS` 值决定的。 缺省值是32。

`segment_size` (`integer`)

报告可以存储在一个文件段中的块（页）数。当构建服务器时， 它是由 `RELSEG_SIZE` 的值决定的。 字节中段文件的最大大小等于 `segment_size` 乘以 `block_size` ；默认情况下为1GB。

`server_encoding` (`string`)

报告数据库编码(字符集)。这是在创建数据库的时候决定的。 通常，客户端只需要关心client_encoding的值。

`server_version` (`string`)

报告服务器版本号。它是由编译服务器时的 `PG_VERSION` 值决定的。

`server_version_num` (`integer`)

报告服务器版本号的整数值。它是由编译服务器时的 `PG_VERSION_NUM` 值决定的。

`wal_block_size` (`integer`)

报告WAL磁盘块大小。当构建服务时，它是通过 `XLOG_BLCKSZ` 的值决定的。 缺省值时8192字节。

`wal_segment_size` (`integer`)

报告在WAL段文件中块（页）数。字节中WAL段文件总的大小等于 `wal_segment_size` 乘以 `wal_block_size` ；缺省是16MB。参见Section 29.4 获取更多详细信息。

18.16. 自定义选项

这个特性用来允许那些由附加模块添加(比如过程语言)的选项， 通常PostgreSQL并不知道它们。这样，扩展的模块就可以用标准的方式配置。

自定义选项有两部分名称：一个扩展名，一个点， 参数名称合适，类似于SQL中限定名称。 例子是 `plpgsql.variable_conflict` 。

因为自定义选项可能需要在没有加载相关扩展模块的过程中进行设置， PostgreSQL将接受任何两部分参数名称的设置。 这样的变量被视为占位符，并没有功能，直到模块定义它们被加载。 当加载一个扩展模块时，这将增加它的变量定义，根据这些定义转换任何占位符值，以及任何以扩展名开头的未确认的占位符发出的警告。

18.17. 开发人员选项

下面的选项目的是在PostgreSQL代码上使用，并且在某些情况下可以帮助恢复严重损坏了的数据库。 在生产环境里没有理由使用这些设置。因此，我们把他们从样例 `postgresql.conf` 文件中排除了出去。 请注意许多这些选项要求特殊的源代码编译标志才能运转。

`allow_system_table_mods` (`boolean`)

允许修改系统表的结构。它可以被 `initdb` 使用。这个值只能在服务器启动的时候设置。

`debug_assertions` (`boolean`)

打开各种断言检查。这是调试助手。如果你经历了奇怪的问题或者崩溃， 那么你可能会想把这个打开，因为它可能暴露编程的错误。要使用这个选项， 我们必须在编译PostgreSQL的时候定义宏 `USE_ASSERT_CHECKING` (通过 `configure` 选项 `--enable-cassert` 完成)。 请注意，如果启用断言选项编译PostgreSQL，那么 `debug_assertions` 缺省就是 `on`

`ignore_system_indexes` (`boolean`)

读取系统表时忽略系统索引(但是修改系统表时依然同时修改索引)。 这个在从系统索引被破坏的表中恢复数据的时候很有用。该参数不能在会话启动之后修改。

`post_auth_delay` (`integer`)

如果为非零，那么在一个新的服务器进程启动并完成认证过程之后， 就会延迟这么多秒。这样就给开发人员一个机会用调试器附着在一个服务器进程上。 该参数不能在会话启动之后修改。

`pre_auth_delay` (`integer`)

如果为非零，那么在一个新的服务器进程派生出来之后， 就会延迟这么多秒，然后才会继续认证过程。 这样就给开发人员一个机会用调试器附着在一个服务器进程上跟踪认证里面的异常行为。 这个选项只能在服务器启动的时候或者在 `postgresql.conf` 文件里设置。

`trace_notify` (`boolean`)

为 `LISTEN` 和 `NOTIFY` 命令生成大量调试输出。 client_min_messages或者log_min_messages必须是 `DEBUG1` 。 或者更低才能把这些输出分别发送到客户端或者服务器日志。

`trace_recovery_messages` (`enum`)

启用恢复相关的调试输出的日志记录，否则 不会被记录。该参数允许用户覆盖log_min_messages的正常设置， 但仅限于特定消息。打算调试双机热备时使用。 有效值 `DEBUG5` ， `DEBUG4` ，`DEBUG3` ， `DEBUG2` ， `DEBUG1` 和 `LOG` 。 默认情况下， `LOG` 不影响记录决

定。其他值会导致恢复相关的记录优先级或更高的调试消息，虽然他们有`LOG`的优先级;对于`log_min_messages`常用的设置导致无条件地发送这些到服务器日志。这个参数只能在`postgresql.conf`文件或者服务器命令行中设置。

`trace_sort` (`boolean`)

如果打开，发出在排序操作中的资源使用的有关信息。这个选项只有在编译PostgreSQL的时候定义了`TRACE_SORT`宏的时候才可用(不过，目前`TRACE_SORT`缺省就是定义了的)。

`trace_locks` (`boolean`)

如果打开，发出关于锁用法信息。信息转储包括锁定操作的类型，锁定类型和被锁定或解锁的对象的唯一标识符。还包括已授权在该对象上的锁类型以及等待此对象上的锁类型的位掩码。对于每个锁类型的授予锁和等待锁的数量计数以及总数都被转储。该日志文件输出的例子在这显示：

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

结构详情可以从`src/include/storage/lock.h`里找到。

当编译PostgreSQL时，如果定义了`LOCK_DEBUG`宏指令，则只能使用这个参数。

`trace_lwlocks` (`boolean`)

如果打开，则发出轻量级锁用法信息。轻量级锁主要提供访问互斥以共享内存数据结构。

当编译PostgreSQL时，如果定义了`LOCK_DEBUG`宏指令，则只能使用这个参数。

`trace_userlocks` (`boolean`)

如果打开，则发出关于用户锁用法的信息。输出类似于`trace_locks`，仅仅为了咨询锁。

当编译PostgreSQL时，如果定义了`LOCK_DEBUG`宏指令，则只能使用这个参数。

`trace_lock_oidmin` (`integer`)

如果设置，不跟踪低于这个OID的表锁。（为了避免系统表输出）

当编译PostgreSQL时，如果定义了`LOCK_DEBUG`宏指令，则只能使用这个参数。

`trace_lock_table` (`integer`)

无条件跟踪表（OID）上的锁。

当编译PostgreSQL时，如果定义了 `LOCK_DEBUG` 宏指令，则只能使用这个参数。

`debug_deadlocks` (`boolean`)

当死锁发生超时，如果设置，那么备份所有当前锁的信息。

当编译PostgreSQL时，如果定义了 `LOCK_DEBUG` 宏指令，则只能使用这个参数。

`log_btree_build_stats` (`boolean`)

如果设置，日志系统资源用法在各种B-tree操作上统计(内存和CPU)。

当编译PostgreSQL时，如果定义了 `BTREE_BUILD_STATS` 宏指令，则只能使用这个参数。

`wal_debug` (`boolean`)

打开 WAL 相关的调试输出。只有在编译PostgreSQL的时候打开了 `WAL_DEBUG` 宏定义的情况下，这个选项才可用。

`ignore_checksum_failure` (`boolean`)

如果启动data checksums，已经受到影响。

在读期间校验失败检测导致PostgreSQL报告错误， 终止当前事务。设置 `ignore_checksum_failure` 的原因系统忽略失败 （但仍然报告一个警告），并且 继续处理。这种行为可能造成死机，传播或隐藏的崩溃，或其他严重的问题。 然而，如果块头仍然是清醒的，它可能允许你获取错误并且检索仍然完好无损出现在表中未处理元组 。 如果头部崩溃，即使启用这个选项也将报告一个错误。 默认设置为 `off` ，它只能由超级用户改变。

`zero_damaged_pages` (`boolean`)

如果侦测到一个损坏了的页面头通常会导致PostgreSQL报告一个错误，并且退出当前事务。把 `zero_damaged_pages` 设置为 on 则令系统报告一个警告，把内存中损坏的页面填充零，然后继续处理。 这种行为会破坏数据，也就是所有在已经损坏页面上的行。 但是它允许你绕开坏页面然后从表中尚存的未损坏页面上继续检索数据行。 因此它在因为硬件或者软件错误导致的崩溃中进行恢复是很有用的。 通常你不应该把它设置为 on ， 除非你已经彻底放弃从崩溃的页面中恢复数据。 零填充页面不强制到磁盘，所以建议重新创建表或再次关闭此参数之前的索引。 缺省的设置是 `off` ，并且只有超级用户可以改变它。

18.18. 短选项

为了方便起见，这里还为一些参数提供了好多单字母命令行选项开关。 它们在下面的Table 18-2里描述。 其中一些选项仅仅是因为历史原因而存在的，尽管它们是单字母选项， 但是并不表示它们很常用。

Table 18-2. 短选项键字

短选项	等效
-A _x_	debug_assertions = _x_
-B _x_	shared_buffers = _x_
-d _x_	log_min_messages = DEBUG``_x_
-e	datestyle = euro
-fb , -fh , -fi , -fm , -fn , -fo , -fs , -ft	enable_bitmapscan = off , enable_hashjoin = off , enable_indexscan = off , enable_mergejoin = off , enable_nestloop = off , enable_indexonlyscan = off , enable_seqscan = off , enable_tidscan = off
-F	fsync = off
-h _x_	listen_addresses = _x_
-i	listen_addresses = '*'
-k _x_	unix_socket_directories = _x_
-l	ssl = on
-N _x_	max_connections = _x_
-O	allow_system_table_mods = on
-p _x_	port = _x_
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S _x_	work_mem = _x_
-tpa , -tpl , -te	log_parser_stats = on , log_planner_stats = on , log_executor_stats = on
-W _x_	post_auth_delay = _x_

Chapter 19. 用户认证

Table of Contents

当客户端与数据库服务器连接时，它将声明以哪个PostgreSQL 数据库用户身份进行连接，就像我们登录一台 Unix 计算机一样。在 SQL 环境里， 活动的数据库用户名决定数据库对象的各种访问权限(参阅 Chapter 20)。 因此，实际上我们要限制的是用户可以连接的数据库。

> **Note:** 如Chapter 20里解释的那样，PostgreSQL 实际上用"角色"的概念管理权限。在本章里，我们用数据库用户表示 "带有 `LOGIN` 权限的角色"。

认证是数据库服务器识别客户端的过程。它通过一些手段判断是否允许此客户端 (或者运行这个客户端的用户)与它所声明的数据库用户名进行绑定。

PostgreSQL提供多种不同的客户端认证方式。这些方式用来认证特定的客户端连接， 可以基于(客户端)的主机地址、数据库、用户选择认证方法。

PostgreSQL数据库用户名在逻辑上是和服务器运行的操作系统用户名相互独立的。 如果某个服务器的所有用户在那台服务器机器上也有帐号，那么给数据库用户赋与操作系统用户名是有意义的。 不过，一个接收远程访问的服务器很有可能有许多没有本地操作系统帐号的用户， 因而在这种情况下数据库用户和操作系统用户名之间不必有任何联系。

19.1. `pg_hba.conf` 文件

客户端认证是由一个配置文件(通常名为 `pg_hba.conf`)控制的， 它存放在数据库集群的数据目录里。HBA的意思是"host-based authentication"， 也就是基于主机的认证。在 `initdb` 初始化数据目录的时候， 它会安装一个缺省的 `pg_hba.conf` 文件。不过我们也可以把认证配置文件放在其它地方； 参阅hba_file配置参数。

`pg_hba.conf` 文件的常用格式是一组记录，每行一条。空白行将被忽略， 井号 `#` 开头的注释也被忽略。记录不能跨行存在。 一条记录是由若干用空格和/或制表符分隔的字段组成。如果字段用引号包围，那么它可以包含空白。 在数据库、用户或地址文件中引用一个关键词（如， `all` 或 `replication` ） 使这个词失去它的特殊角色，只是用这个名字匹配一个数据库、用户或主机。

每条记录声明一种连接类型、一个客户端 IP 地址范围(如果和连接类型相关的话)、 一个数据库名、一个用户名字、对匹配这些参数的连接使用的认证方法。第一条匹配连接类型、 客户端地址、连接请求的数据库名和用户名的记录将用于执行认证。这个处理过程没有 "跨越"或者"回头"的说法：如果选择了一条记录而且认证失败， 那么将不再考虑后面的记录。如果没有匹配的记录，那么访问将被拒绝。

每条记录可以是下面七种格式之一：

```
local      _database_  _user_  _auth-method_  [`_auth-options_`]
host       _database_  _user_  _address_  _auth-method_  [`_auth-options_`]
hostssl    _database_  _user_  _address_  _auth-method_  [`_auth-options_`]
hostnossl  _database_  _user_  _address_  _auth-method_  [`_auth-options_`]
host       _database_  _user_  _IP-address_  _IP-mask_  _auth-method_  [`_auth-options_`]
hostssl    _database_  _user_  _IP-address_  _IP-mask_  _auth-method_  [`_auth-options_`]
hostnossl  _database_  _user_  _IP-address_  _IP-mask_  _auth-method_  [`_auth-options_`]
```

各个字段的含义如下：

`local`

这条记录匹配企图通过 Unix 域套接字进行的连接。没有这种类型的记录，就不允许 Unix 域套接字的连接。

`host`

这条记录匹配企图通过 TCP/IP 进行的连接。 `host` 记录匹配 SSL和非SSL的连接请求。

> **Note:** 除非服务器带着合适的listen_addresses配置参数值启动， 否则将不可能进行远程的 TCP/IP 连接，因为缺省的行为是只监听本地自环地址 `localhost` 的连接。

`hostssl`

这条记录匹配企图使用 TCP/IP 的 SSL 连接。但必须是使用SSL加密的连接。

要使用这个选项，编译服务器的时候必须打开SSL支持。 而且在服务器启动的时候必须打开ssl配置选项(参阅Section 17.9)。

`hostnossl`

这条记录与 `hostssl` 行为相反：它只匹配那些在 TCP/IP 上不使用SSL的连接请求。

`_database_`

声明记录所匹配的数据库名称。值 `all` 表明该记录匹配所有数据库， 值 `sameuser` 表示如果被请求的数据库和请求的用户同名，则匹配。 值 `samerole` 表示请求的用户必须是一个与数据库同名的角色中的成员。 (`samegroup` 是一个已经废弃了，但目前仍然被接受的 `samerole` 同义词。) 对 `samerole` 来说，不认为超级用户是角色的一个成员，除非他们明确的是角色的成员， 直接的或间接的，并且不只是由于超级用户。值 `replication` 表示如果请求一个复制链接， 则匹配（注意复制链接不表示任何特定的数据库）。在其它情况里，这就是一个特定的PostgreSQL数据库名字。可以通过用逗号分隔的方法声明多个数据库， 也可以通过前缀 `@` 来声明一个包含数据库名的文件。

`_user_`

为这条记录声明所匹配的数据库用户。值 `all` 表明它匹配于所有用户。否则， 它就是特定数据库用户的名字或者是一个前缀 `+` 的组名称。请注意， 在PostgreSQL里，用户和组没有真正的区别， `+` 实际上只是意味着 "匹配任何直接或者间接属于这个角色的成员"，而没有 `+` 记号的名字只匹配指定的角色。 为此，超级用户如果明确是角色的成员，也只算是一个角色的成员，直接的或间接的，而不只是由于超级用户。 多个用户名可以通过用逗号分隔的方法声明。一个包含用户名的文件可以通过在文件名前面前缀 `@` 来声明。

`_address_`

声明这条记录匹配的客户端机器地址。这个文件可以包含主机名、IP地址范围或下面提到的特殊关键字之一。

IP 地址用标准的带有CIDR掩码长度的点分十进制声明。 掩码长度表示客户端 IP 地址必须匹配的高位二进制位数。在给出的 IP 地址里， 这个长度的右边的二进制位应该为零。在 IP 地址、 `/` 、 CIDR 掩码长度之间不能有空白。

典型的这种方式指定的IP地址范围举例： `172.20.143.89/32` 表示一个主机， `172.20.143.0/24` 表示一个小子网， `10.6.0.0/16` 表示一个大子网。 `0.0.0.0/0` 代表所有IPv4地址， `::/0` 代表所有IPv6地址。要声明单个主机，给 IPv4 地址声明 CIDR 掩码 32 ，给 IPv6 地址声明 128 。 不要在地址中省略结尾的 0 。

以 IPv4 格式给出的 IP 地址会匹配那些拥有对应地址的 IPv6 连接，比如 `127.0.0.1` 将匹配 IPv6 地址 `::ffff:127.0.0.1` 。一个以 IPv6 格式给出的记录将只匹配 IPv6 连接， 即使对应的地址在 IPv4-in-IPv6 范围内。请注意如果系统的 C 库不支持 IPv6 地址， 那么 IPv6 的格式将被拒绝。

你也可以写 `all` 来匹配所有IP地址， `samehost` 来匹配任意服务器IP地址，或 `samenet` 来匹配任何服务器直接连接到的子网的任意地址。

如果指定了主机名（不是IP地址或作为潜在主机名处理的特殊关键字）， 那么该名称与客户端IP地址进行反向名称解析的结果进行比较（例如，如果使用了DNS， 那么是反向DNS查找）。主机名的比较是大小写无关的。如果有一个匹配， 那么正向名称解析（例如，正向DNS查找）在主机名上执行，以检查是否有解析的地址等于客户端IP地址。 如果双向都匹配，那么这个条目被认为是匹配的。（在 `pg_hba.conf` 中使用的主机名应该是客户端IP地址返回的地址到名称（address-to-name）解析的那个， 否则这行将不被匹配。某些主机名数据库允许一个IP地址关联多个主机名， 但是当要求解析一个IP地址时，操作系统将只返回一个主机名。）

主机名规范以一个点(`.`)开头，匹配一个实际主机名后缀。 所以 `.example.com` 将匹配 `foo.example.com` （但不只是 `example.com` ）。

当主机名在 `pg_hba.conf` 中指定时，你应该确保那个名字解析是相当快的。 它将比建立一个本地名字解析缓存（如 `nscd` ）有优势。同样， 你可能希望启用配置参数 `log_hostname` 来查看客户端主机名，而不是日志中的IP地址。

有时，用户想知道为什么主机名用这个带有两个名字解析和要求反向查找IP地址的看起来复杂的方式处理， 这有时没有设置或者指向一些不受欢迎的主机名。效率是首要的：一个连接请求需要两个解析器查找当前客户端地址。 如果那个地址有解析器问题，那么就只是客户端问题。一个假想的可供选择的只做正向查找的实现， 将在每个连接请求时必须解析 `pg_hba.conf` 中提到的每个主机名。这样本身就是缓慢的。 并且如果有一个主机名有解析问题，那么所有主机名都会有问题。

另外，要实现后缀匹配功能必须要一个反向查找，因为实际客户端主机名需要是已知的，为了它对模式匹配。

请注意，这个行为与其他受欢迎的主机基于名称访问控制的实现是一致的，比如Apache HTTP Server 和 TCP Wrappers。

这些字段只适用于 `host` , `hostssl` , `hostnossl` 记录。

`_IP-address_``_IP-mask_`

这些方法可以用于作为 `_CIDR-address_` 表示法的替补。它不是声明掩码的长度， 而是在另外一个字段里声明实际的掩码。比如， `255.0.0.0` 表示 IPv4 CIDR 掩码长度 8 ，而 `255.255.255.255` 表示 CIDR 掩码长度 32 。

这些字段只适用于 `host` , `hostssl` , `hostnossl` 记录。

`_auth-method_`

声明连接匹配这条记录的时候使用的认证方法。可能的选择在下面简介， 详细情况在Section 19.3中介绍。

`trust`

无条件地允许连接。这个方法允许任何可以与PostgreSQL 数据库服务器连接的用户以他们期望的任意PostgreSQL 数据库用户身份进行连接，而不需要口令或任何其他认证。参阅Section 19.3.1获取细节。

`reject`

无条件地拒绝连接。常用于从一个组中"过滤"某些主机，例如， 一个 `拒绝` 行能够从连接中锁定一个指定的主机，而稍后的行允许指定网络中的剩余的主机连接。

`md5`

要求客户端提供一个 MD5 加密的口令进行认证。参阅Section 19.3.2获取细节。

`password`

要求客户端提供一个未加密的口令进行认证。因为口令是以明文形式在网络上传递的， 所以我们不应该在不安全的网络上使用这个方式。参阅Section 19.3.2获取细节。

`gss`

使用GSSAPI认证用户。这只能用于TCP/IP连接。参阅Section 19.3.3获取细节。

`sspi`

使用SSPI认证用户。这只能在Windows上使用。参阅Section 19.3.4获取细节。

`krb5`

用 Kerberos V5 认证用户。只有在进行 TCP/IP 连接的时候才能用。 参阅Section 19.3.5获取细节。

`ident`

获取客户的操作系统名然后检查该用户是否匹配要求的数据库用户名， 方法是用户的身份通过与运行在客户端上的 ident 服务器连接进行判断的。 Ident认证只在进行TCP/IP连接的时候才能用。当指定本地连接时，将使用peer认证。 参阅Section 19.3.6获取细节。

`peer`

为操作系统获取客户端操作系统用户名，并检查该用户是否匹配要求的数据库用户名。 该方法只适用于本地连接。参阅Section 19.3.7获取细节。

`ldap`

使用LDAP服务器进行认证。参阅Section 19.3.8获取细节。

`radius`

使用RADIUS服务器进行认证，参阅Section 19.3.9获取细节。

`cert`

使用SSL客户端证书进行认证。参阅Section 19.3.10获取细节。

`pam`

使用操作系统提供的可插入认证模块服务(PAM)来认证。参阅Section 19.3.11获取细节。

`_auth-options_`

在 `_auth-method_` 字段之后，字段格式可以是 `_name_``=``_value_`， 指定认证方法的选项。关于哪个选项可用于哪个认证方法的详情在下面描述。

用 `@` 构造包含的文件是当作一列名字读取的，这些名字可以用空白或者逗号分隔。 注释用 `#` 引入，就像在 `pg_hba.conf` 里那样， 允许嵌套 `@` 构造。除非跟在 `@` 后面的文件名是一个绝对路径， 否则被当作与该文件所在目录相对的路径。

因为认证时系统是为每个连接请求顺序检查 `pg_hba.conf` 里的记录的， 所以这些记录的顺序是非常关键的。通常，靠前的记录有比较严的连接匹配参数和比较弱的认证方法， 而靠后的记录有比较松的匹配参数和比较严的认证方法。比如，我们一般都希望对本地 TCP/IP 连接使用 `trust` 认证，而对远端的 TCP/IP 连接要求口令。在这种情况下我们将 `trust` 认证方法用于来自 127.0.0.1 的连接，这条记录将出现在允许更广泛的客户端 IP 地址的使用口令认证的记录前面。

在启动和主服务器进程收到SIGHUP 信号的时候，系统都会重新装载 `pg_hba.conf` 文件。 如果你在活跃的系统上编辑了该文件，就必须通知主服务器(使用 `pg_ctl reload` 或 `kill -HUP`)重新加载该文件。

> **Tip:** 一个用户要想成功连接到特定的数据库，不仅需要通过 `pg_hba.conf` 的检查， 还必须要有该数据库上的 `CONNECT` 权限。如果希望限制哪些用户能够连接到哪些数据库， 赋予/撤销 `CONNECT` 权限通常比在 `pg_hba.conf` 中设置规则简单。

Example 19-1里是 `pg_hba.conf` 记录的一些例子。 阅读下文理解不同认证方法的细节。

Example 19-1. `pg_hba.conf` 记录的例子

```
# 允许在本机上的任何用户使用 Unix 域套接字(本地连接的缺省)
# 以任何数据库用户身份连接任何数据库
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# 和上面相同，但是使用的是回环的(loopback)TCP/IP 连接
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# 和上面一行相同，但是用的是独立的子网掩码字段
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# 在IPv6上相同。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust
```

```
# 和上面相同，但是使用一个主机名（通常包括IPv4 和 IPv6）。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# 允许 IP 地址为 192.168.93.x 的任何主机与 "postgres" 数据库相连，
# 用与他们在自己的主机上相同 ident 的用户名标识他自己(通常是他的操作系统用户名)
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident

# 允许来自主机 192.168.12.10 的用户提供了正确的口令之后与 "postgres" 数据库连接。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        md5

# 允许来自在example.com域里的主机的用户在提供了正确的口令之后与任意数据库连接。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             .example.com            md5

# 如果前面没有其它 "host" 行，那么下面两行将拒绝所有来自 192.168.54.1 的连接请求(因为前面的记录先匹配)
# 但是允许来自互联网上其它任何地方的有效的 Kerberos 5 认证的连接。
# 零掩码引起不考虑主机 IP 的任何位。因此它匹配任何主机。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               krb5

# 允许来自 192.168.x.x 的任何用户与任意数据库连接，只要他们通过 ident 检查。
# 但如果 ident 说该用户是 "bryanh" 且他要求以 PostgreSQL 用户 "guest1" 连接，
# 那么只有在 pg_ident.conf 里有 "omicron" 的映射说 "bryanh" 允许以 "guest1" 进行连接时才真正可以进
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron

# 如果下面是用于本地连接的仅有的三行，那么它们将允许本地用户只和同名数据库连接。
# 只有管理员和 "support" 角色里的成员例外，他们可以连接到任何数据库。
# $PGDATA/admins 文件列出了那些允许与所有数据库连接的用户名。
# 在所有情况下都需要口令。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# 上面最后两行可以合起来写成一行
local   all             @admins,+support                        md5

# 数据库字段也可以使用列表和文件名：
local   db1,db2,@demodbs  all                                   md5
```

19.2. 用户名映射

当使用像Ident 或 GSSAPI这样的外部认证系统时，发起连接的操作系统用户名可能与他需要连接的数据库用户是不同的。 在这种情况下，用户名映射可用于映射操作系统用户名到数据库用户。要使用用户名映射，在 `pg_hba.conf` 中的选项中指定 `map` = `_map-name_` 。这个选项支持所有接受外部用户名的认证方法。 因为不同的映射可能需要不同的连接，使用的映射名在 `pg_hba.conf` 中的 `_map-name_` 参数中指定，表示为每个独立的连接使用哪个映射。

用户名映射在身份映射文件中定义，缺省名为 `pg_ident.conf` 并且缺省存储在集群的数据目录中。（不过，我们也可以把映射文件放在其它地方；参阅ident_file配置参数。） 身份映射文件包含的下面通用的格式：

```
_map-name_ _system-username_ _database-username_
```

注释和空白与 `pg_hba.conf` 文件里的一样处理。 `_map-name_` 是将要用于在 `pg_hba.conf` 中引用这个映射的任意名称。 另外两个字段声明操作系统用户名和匹配的数据库用户名。同一个 `_map-name_` 可以在一个映射中多次使用来声明多个用户映射。 对一个操作系统用户可以映射为多少个数据库用户没有限制，反之亦然。

对一个给定的操作系统用户可以映射为多少个数据库用户没有限制，反之亦然。 因此，映射中的记录应该被认为是意为"这个操作系统用户被允许作为这个数据库用户连接"， 而不是意味着他们是相等的。如果有那对用户名的任意映射记录从带有那个用户请求连接的数据库用户名的外部认证系统获得， 那么这个连接将被允许。

如果 `_system-username_` 字段以一个反斜杠(`/`)开始， 那么该字段的剩余部分被作为一个规则表达式对待。（参阅Section 9.7.3.1 获取关于PostgreSQL规则表达式语法的细节。）规则表达式可以包含一个捕获， 或括号的子表达式，然后可以在 `_database-username_` 字段中作为 `\1` (backslash-one)引用。 这允许多用户名的映射在一行中，这对于简单的语法替换特别有用。例如，这些记录：

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

将为带有系统用户名的以 `@mydomain.com` 结尾的用户删除域部分， 并且允许系统名以 `@otherdomain.com` 结尾的任意用户作为 `guest` 登录。

> **Tip:** 记住，缺省的，一个规则表达式只可以匹配字符串的一部分。使用 `^` 和 `$` 通常是明智的， 就像上面例子所示，来迫使匹配成为整个系统用户名。

在系统启动和主服务器收到一个SIGHUP 信号的时候会读取 `pg_ident.conf` 文件。如果你在一台活跃的系统上编辑该文件， 那么你需要给主服务器发信号(使用 `pg_ctl reload` 或 `kill -HUP`)令其重新读取该文件。

Example 19-2里是一个可以和在Example 19-1 里面演示的 `pg_hba.conf` 文件配合使用的 `pg_ident.conf` 文件。 在这个例子的设置里，任何登录到 192.168 网络里的机器的用户，如果用户名不是 `bryanh` ， `ann` ， `robert` 就不能获准访问。Unix 用户 `robert` 只有在试图以PostgreSQL用户 `bob` 身份连接时才允许访问，而不能是 `robert` 或其它什么身份。 `ann` 将只允许以 `ann` 的身份连接。用户 `bryanh` 允许以他自己的 `bryanh` 身份或者作为 `guest1` 进行连接。

Example 19-2. 一个 `pg_ident.conf` 文件例子

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# bob 在这台机器上的用户名是 robert
omicron         robert                  bob
# bryanh也可以以 guest1 身份连接
omicron         bryanh                  guest1
```

19.3. 认证方法

下面的小节更详细地描述认证方法。

19.3.1. 信任认证

如果声明了 `trust` 认证模式，PostgreSQL 就假设任何可以连接到服务器的人都可以以任何他声明的数据库用户名(甚至超级用户名)连接。 当然，在 `database` 和 `user` 字段里面的限制仍然适用。 这个方法应该用于那些在连接到服务器已经有足够操作系统层次保护的环境里。

`trust` 认证对于单用户工作站的本地连接是非常合适和方便的。 通常它本身并不适用于多用户环境的机器。不过，即使在多用户的机器上， 你也可以使用 `trust` ，只要你利用文件系统权限限制了对服务器的 Unix 域套接字文件的访问。 要做这些限制，你可以设置 `unix_socket_permissions` 参数(以及可能还有 `unix_socket_group`)，就像Section 18.3里描述的那样。 或者你可以设置 `unix_socket_directories` ，把 Unix 域套接字文件放在一个经过恰当限制的目录里。

设置文件系统权限只能帮助 Unix 套接字连接，它不会限制本地 TCP/IP 连接。因此， 如果你想利用文件系统权限来控制本地安全，那么删除 `pg_hba.conf` 文件中的 `host ... 127.0.0.1 ...` 行，或者把它改为一个非 `trust` 的认证方法。

`trust` 认证模式只适合 TCP/IP 连接，只有在你信任那些 `trust` 行上所有机器中的所有用户的时候才是合适的。 很少有理由使用 `trust` 作为任何除来自localhost (127.0.0.1) 以外的 TCP/IP 连接的认证方式。

19.3.2. 口令认证

以口令为基础的认证方法包括 `md5` , `password` 。这些方法操作上非常类似， 只不过口令通过连接传送的方法不同：分别是MD5 散列、明文。

如果你担心口令被"窃听"，那么 `md5` 比较合适。 应该尽可能避免使用 `password` 。然而， `md5` 不能和 db_user_namespace功能一起使用。如果连接通过SSL加密保护， 那么 `password` 可以安全的使用（尽管如果一个用户依赖于使用SSL， SSL证书认证可能是一个更好的选择。）

PostgreSQL数据库口令与任何操作系统用户口令无关。 各个数据库用户的口令是存储在 `pg_authid` 系统表里面。 口令可以用 SQL 语言命令CREATE USER和ALTER ROLE 等管理(比如<kbd class="literal">CREATE USER foo WITH PASSWORD 'secret'</kbd>。 如果没有明确设置口令，那么存储的口令是空并且该用户的口令认证总会失败。

19.3.3. GSSAPI 认证

GSSAPI是为了在RFC 2743中定义的安全认证的一个工业标准协议。 PostgreSQL根据RFC 1964支持GSSAPI 和Kerberos认证一起使用。GSSAPI 为支持它的系统提供自动身份验证（单点登录）。身份验证本身是安全的， 但是数据在数据连接时的传送将会是未加密的，除非使用了SSL。

当GSSAPI使用Kerberos时，它使用一个标准， 主要以 `_servicename_` / `_hostname_` @ `_realm_` 的格式。 关于主要部分的信息和如何配置所需的秘钥，请参阅Section 19.3.5。

GSSAPI支持在PostgreSQL编译时必须打开；参阅Chapter 15获取更多信息。

GSSAPI支持下列的配置选项：

`include_realm`

如果设置为1，通过身份验证的用户主的域名包含在通过用户名映射的系统用户名中(Section 19.2)。 这对处理来自多个领域的用户是有帮助的。

`map`

允许在系统和数据库用户名之间映射。参阅Section 19.2获取细节。 对于一个Kerberos主要的 `username/hostbased@EXAMPLE.COM` ， 如果 `include_realm` 未启用，那么用来映射的用户名是 `username/hostbased` ， 如果 `include_realm` 启用了，那么就是 `username/hostbased@EXAMPLE.COM` 。

`krb_realm`

设置域以匹配用户主名称。如果设置了这个参数，那么只接受那个域中的用户。如果没有设置， 那么任何域中的用户都可以连接，使无论什么用户名映射都完成。

19.3.4. SSPI 认证

SSPI是单点登录安全身份验证的一个Windows技术。 PostgreSQL将在 `negotiate` 模式下使用SSPI， 当可能时将使用Kerberos并且在其他情况下会自动回滚至NTLM。 SSPI认证只当服务器和客户端都运行Windows时工作， 否则，在非Windows平台上，GSSAPI是可用的。

当使用Kerberos认证时，SSPI以和GSSAPI 一样的方式工作；参阅Section 19.3.3获取详细信息。

SSPI支持下列的配置选项：

`include_realm`

如果设置为1，通过身份验证的用户主的域名包含在通过用户名映射的系统用户名中(Section 19.2)。 这对处理来自多个领域的用户是有帮助的。

`map`

允许在系统和数据库用户名之间映射。参阅Section 19.2获取细节。

`krb_realm`

设置域以匹配用户主名称。如果设置了这个参数，那么只接受那个域中的用户。如果没有设置， 那么任何域中的用户都可以连接，使无论什么用户名映射都完成。

19.3.5. Kerberos 认证

> **Note:** 本地Kerberos认证已经废弃了而且应该只在为了向后兼容的时候使用。 建议新的和升级安装使用工业标准GSSAPI认证方法（参阅Section 19.3.3）。

Kerberos是一种适用于在公共网络上进行分布计算的工业标准的安全认证系统。 对Kerberos系统的叙述超出了本文档的范围； 总的说来它是相当复杂(同样也相当强大)的系统。Kerberos FAQ或MIT Kerberos page 是个开始学习的好地方。现存在好几种Kerberos发布的源代码。Kerberos 只提供安全认证，但并不加密在网络上传输的查询和数据，SSL可以用于这个目的。

PostgreSQL支持 Kerberos 5 ， Kerberos 支持必须在编译的时候打开。 参阅Chapter 15获取更多信息。

PostgreSQL运行时像一个普通的 Kerberos 服务。服务主的名字是 `_servicename_` / `_hostname_` @ `_realm_` 。

`_servicename_` 可以用krb_srvname配置参数在服务器端设置， 或者在客户端使用 `krbsrvname` 连接参数设置(又见Section 31.1.2)。 编译的时候，可以把安装时的缺省 `postgres` 修改掉，方法是使用 `./configure --with-krb-srvname=``_whatever_` 。在大多数情况下， 我们不需要修改这个参数。但是，如果需要在同一台主机上同时安装多套PostgreSQL， 那么这个就是必须的了。有些 Kerberos 实现还可能要求其它的服务名， 比如Microsoft Active Directory 就要求服务名必须是大写的( `POSTGRES`)。

`_hostname_` 是服务器的全限定主机名。服务主的领域就是主机的首选领域。

客户主自己必须用它们自己的PostgreSQL用户名作为第一个部件， 比如 `pgusername@realm` 。或者，你可以使用一个用户名映射来从主名的第一个部件到数据库用户名映射。 缺省的，PostgreSQL没有检查客户的域；因此如果你打开了跨域的认证，并且需要验证这个域， 那么使用 `krb_realm` 参数或者打开 `include_realm` 并使用用户名映射来检查这个域。

确认服务器的密钥表文件是可以被PostgreSQL服务器帐户读取 (最好就是只读的)。(又见Section 17.1。)密钥文件(keytab)的位置是用配置参数 krb_server_keyfile声明的。缺省是 `/usr/local/pgsql/etc/krb5.keytab` (或者任何在编译的时候声明为 `sysconfdir` 的目录)。

密钥表文件(keytab)是在 Kerberos 软件里生成的，参阅 Kerberos 文档获取细节。 下面的例子是可以用于 MIT 兼容的 Kerberos 5 实现：

```
<samp class="literal">kadmin%</samp> <kbd class="literal">ank -randkey postgres/server.my
<samp class="literal">kadmin%</samp> <kbd class="literal">ktadd -k krb5.keytab postgres/s
```

在和数据库连接的时候，请确保自己对每个主都拥有一张匹配所请求的数据库用户名的门票。 比如，对于数据库用户 `fred` ，主 `fred@EXAMPLE.COM` 将能够连接。 为了也允许主 `fred/users.example.com@EXAMPLE.COM` ，使用一个用户名映射， 在Section 19.2中描述。

如果你在Apache服务器上使用了 mod_auth_kerb 和mod_perl模块，你可以用一个mod_perl脚本进行 `AuthType KerberosV5SaveCredentials` 。 这样就有了一个通过 web 的安全数据库访问，不需要额外的口令。

Kerberos支持下列的配置选项：

`map`

允许系统和数据库用户名之间的映射。参阅Section 19.2获取详细信息。

`include_realm`

如果设置为1，通过身份验证的用户主的域名包含在通过用户名映射的系统用户名中(Section 19.2)。 这对处理来自多个领域的用户是有帮助的。

`krb_realm`

设置域以匹配用户主名称。如果设置了这个参数，那么只接受那个域中的用户。如果没有设置， 那么任何域中的用户都可以连接，使无论什么用户名映射都完成。

`krb_server_hostname`

设置服务主的主机名部分。与 `krb_srvname` 组合，用来产生全部的服务主，也就是 `krb_srvname``/``krb_server_hostname``@` 域。 如果没有设置，缺省为服务器主机名。

19.3.6. Ident 认证

ident 认证方法是通过从一个ident服务器获取客户端的操作系统用户名， 并使用它作为允许的数据库用户名（带有一个可选的用户名映射）。 只在TCP/IP连接上支持。

> **Note:** 当ident指定为本地连接（非TCP/IP）时，将使用peer的认证（参阅Section 19.3.7）。

ident支持下列的配置选项：

`map`

允许在系统和数据库用户名之间映射。参阅Section 19.2获取详细信息。

"Identification Protocol"(标识协议)在 RFC 1413 里面描述。 实际上每个类 Unix 的操作系统都带着一个缺省时侦听 113 端口的身份服务器。 身份服务器的基本功能是回答类似这样的问题："是什么用户从你的端口 `_X_` 初始化出来连接到我的端口 `_Y_` 上来了?"。 因为在建立起物理连接后，PostgreSQL既知道 `_X_` 也知道 `_Y_`， 因此它可以询问运行尝试连接的客户端的主机，并且理论上可以判断发起连接的操作系统用户。

这样做的缺点是它取决于客户端的完整性：如果客户端不可信或者被盗用， 那么攻击者可以在 113 端口上运行任何程序并且返回他们选择的任何用户。 这个认证方法只适用于封闭的网络， 这样的网络里的每台客户机都处于严密的控制下并且数据库和操作系统管理员可以比较方便地联系上。 换句话说，你必须信任运行身份(ident)服务的机器。下面是警告：

身份标识协议并不适用于认证或者访问控制协议。

--RFC 1413

有些身份服务器有一个非标准的选项，导致返回的用户名是加密的， 使用的是只有原机器的管理员知道的一个密钥。在与PostgreSQL配合使用身份认证的时候， 你一定不能使用这个选项，因为PostgreSQL 没有任何方法对返回的字符串进行解密以获取实际的用户名。

19.3.7. Peer 认证

peer认证方法通过从内核获得客户端的操作系统用户名和使用它作为允许的数据库用户名（使用可选的用户名映射）工作。这个方法只支持本地连接。

peer支持下列的配置选项：

`map`

允许在系统和数据库用户名之间映射。参阅Section 19.2获取详细信息。

Peer认证只能在提供 `getpeereid()` 函数、 `SO_PEERCRED` 套接字参数或类似的机制的操作系统上适用，目前包括Linux、 大多数包含Mac OS X的BSD 和Solaris。

19.3.8. LDAP 认证

这个认证方法操作起来类似 `password` ，只不过它使用 LDAP 作为密码验证机制。 LDAP 只用于验证用户名/口令对。因此，在使用 LDAP 进行认证之前，用户必须已经存在于数据库里。

LDAP认证可以在两种模式操作。在第一种模式，我们将调用简单的绑定模式， 服务器将绑定到以 `_prefix_` `_username_` `_suffix_` 构造的识别名(Distinguished Name)上。通常 `_prefix_` 参数用于在活动目录环境中指定 `cn=` 或 `_DOMAIN_``\` 。 `_suffix_` 用来在非活动目录环境中指定DN的剩余部分。

在第二种模式中，我们将调用搜索+绑定模式，服务器首先绑定到LDAP目录上，带有固定用户名和密码， 用 `_ldapbinddn_` 和 `_ldapbindpasswd_` 指定，并且为试图登陆到数据库的用户执行一次搜索。 如果没有配置用户名和密码，将试图对这个目录进行匿名绑定。将对在 `_ldapbasedn_` 的子树执行搜索，并且将试图对 `_ldapsearchattribute_` 指定的属性做一个准确匹配。 一旦在这个目录中发现了用户，服务器断开并且重新作为这个用户绑定到目录，使用客户端指定的密码， 以验证登陆是正确的。这个模式和在其他软件使用的LDAP认证模式相同，例如Apache mod_authnz_ldap 和 pam_ldap。 这种方法允许在目录中的用户对象有很大的灵活性，但是将导致两个独立的连接到LDAP服务器。

下列的配置选项在两种模式下都使用：

`ldapserver`

连接到的LDAP服务器的名字或IP地址。可能指定多个服务器，用空格分开。

`ldapport`

连接到的LDAP服务器的端口号。如果没有指定端口，将使用LDAP库缺省端口。

`ldaptls`

设置为1以使PostgreSQL和LDAP服务器之间的连接使用TLS加密。注意这只加密去往LDAP服务器的流量， 也就是连接到客户端将仍然是未加密的，除非使用了SSL。

下列的选项只在简单绑定模式中使用：

`ldapprefix`

当做简单的绑定认证，绑定到DN时，前缀到用户名的字符串。

`ldapsuffix`

当做简单的绑定认证，绑定到DN时，附加到用户名的字符串。

下列的选项只在搜索+绑定模式中使用：

`ldapbasedn`

当做搜索+绑定认证时，为用户开始搜索的根DN。

`ldapbinddn`

当做搜索+绑定认证时，绑定到目录并执行搜索的用户的DN。

`ldapbindpasswd`

当做搜索+绑定认证时，绑定到目录并执行搜索的用户的密码。

`ldapsearchattribute`

当做搜索+绑定认证时，在搜索中匹配用户名的属性。如果没有指定属性，将使用 `uid` 属性。

`ldapurl`

一个RFC 4516 LDAP URL。这是一个在更紧凑和标准的形式中写入一些其他LDAP选项的可选择的方式。 格式是

```
ldap://_host_[:_port_]/_basedn_[?[_attribute_][?[_scope_]]]
```

`_scope_` 必须是 `base` , `one` , `sub` 之一，通常是最后一个。只使用一个属性， 一些其他的标准LDAP URL的组成部分比如filters和extensions是不支持的。

对于非匿名的绑定， `ldapbinddn` 和 `ldapbindpasswd` 必须作为独立的选项声明。

要使用加密的LDAP连接，除了 `ldapurl` ， 必须使用 `ldaptls` 选项。 不支持 `ldaps` URL模式(直接SSL连接）。

LDAP URL当前只支持OpenLDAP，不是在Windows上。

混合简单绑定和搜索+绑定的配置选项是错误的。

这里是简单绑定LDAP配置的一个例子：

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=n
```

当要求一个到数据库服务器的连接作为数据库用户 `someuser` 时， PostgreSQL将试图使用DN `cn=someuser, dc=example, dc=net` 绑定到LDAP服务器， 并且密码由客户端提供。如果那个连接成功了，那么就同意数据库访问。

这里是搜索+绑定配置的一个例子：

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchattri
```

当要求一个到数据库服务器的连接作为数据库用户 `someuser` 时， PostgreSQL将试图匿名（因为没有指定 `ldapbinddn` ）绑定到LDAP服务器， 为 `(uid=someuser)` 在指定的基础DN下执行一个搜索。如果发现一条记录， 它将试图使用发现的信息和客户端提供的密码绑定。如果第二个链接成功，那么就同意数据库访问。

这是相同的搜索+绑定配置，写作一个URL：

```
host ... ldap lapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

一些其他支持LDAP认证的软件使用相同的URL格式，所以它将更容易共享配置。

> **Tip:** 因为LDAP经常使用逗号和空格来分开DN的不同部分，所以当配置LDAP选项时，通常需要使用双引号参数值，就像例子中所示的一样。

19.3.9. RADIUS 认证

这个认证方法操作起来类似 `password`，只不过它使用RADIUS作为密码验证方法。 RADIUS只用于验证用户名/口令对。因此，在使用RADIUS进行认证之前，用户必须已经存在于数据库里。

当使用RADIUS认证时，一个访问请求信息将发送到已配置好的RADIUS服务器。 这个请求将会是类型 `Authenticate Only`，并且包含参数 `user name`，`password` (加密的) 和 `NAS Identifier` 。 这个请求将使用一个和服务器秘密共享的加密。RADIUS服务器将以 `Access Accept` 或 `Access Reject` 响应这个服务器。不支持RADIUS账户。

RADIUS支持下列的配置选项：

`radiusserver`

连接到的RADIUS服务器的名字或IP地址。这个参数是必需的。

`radiussecret`

当安全的和RADIUS服务器对话时使用的共享秘钥。在PostgreSQL和RADIUS服务器上必须有完全相同的值。 建议至少为16个字符的字符串。这个参数是必需的。

> **Note:** 如果PostgreSQL编译支持OpenSSL，那么加密向量将只被强大的加密使用。 在其他情况下，到RADIUS服务器的传输应该只被认为是混淆的，不是安全的，并且必要时应该采用外部安全措施。

`radiusport`

连接到的RADIUS服务器的端口号。如果没有指定端口，将使用缺省的端口 `1812` 。

`radiusidentifier`

在RADIUS请求中作为 `NAS Identifier` 使用的字符串。这个参数可以用作第二个参数标识，比如用户试图作为哪个数据库用户验证，哪个可以用来在RADIUS服务器上政策匹配。 如果没有指定标识符，将使用缺省的 `postgresql` 。

19.3.10. 证书认证

这个认证方法使用SSL证书来执行认证。因此只适用于SSL连接。当使用这个认证方法时，服务器将请求客户端提供一个有效的证书。没有密码提示发送给客户端。 证书的 `cn` (Common Name) 属性将与请求的数据库用户名比较， 如果它们匹配，登陆将被允许。用户名映射可以用来允许 `cn` 不同于数据库用户名。

SSL证书认证支持下列的配置选项：

`map`

允许在系统和数据库用户名之间映射。参阅Section 19.2获取详细信息。

19.3.11. PAM 认证

这个认证方法操作起来类似`password`，只不过它使用 PAM (Pluggable Authentication Modules)作为认证机制。缺省的 PAM 服务名是`postgresql`。 PAM 只用于验证用户名/口令对。因此，在使用 PAM 进行认证之前，用户必须已经存在于数据库里。 有关 PAM 的更多信息，请阅读 Linux-PAM页面。

PAM支持下列的配置选项：

`pamservice`

PAM服务名。

> **Note:** 如果PAM设置为读取`/etc/shadow`，那么将验证失败，因为PostgreSQL服务器是通过非root用户启动的。 然而，当PAM设置为使用LDAP或其他认证方法时将不是一个问题。

19.4. 用户认证

认证失败以及相关的问题通常由类似下面的错误信息表白。

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

这条信息出现的最大可能是你已经连接了服务器，但它不愿意和你说话。 就像信息自己表示的那样，服务器拒绝了连接请求，因为它没有在它的 `pg_hba.conf` 配置文件里找到匹配的记录。

```
FATAL:  password authentication failed for user "andym"
```

这样的信息表示你连接了服务器，并且它也愿意和你交谈，但是你必须通过 `pg_hba.conf` 文件里声明的认证方法。检查你提交的口令， 或者如果错误信息提到这些 Kerberos 或 IDENT 认证类型时检查这些软件。

```
FATAL:  user "andym" does not exist
```

这是表示数据库用户不存在的。

```
FATAL:  database "testdb" does not exist
```

你试图连接的数据库不存在。请注意如果你没有声明数据库名，缺省是数据库用户名， 这可能正确也可能不正确。

> **Tip:** 请注意服务器日志可能包含比报告给客户端的更多的有关认证失败的信息。 如果你被失败的原因搞糊涂了，那么请检查服务器日志。

Chapter 20. 数据库角色

Table of Contents

PostgreSQL使用角色的概念管理数据库访问权限。 根据角色自身的设置不同，一个角色可以看做是一个数据库用户，或者一组数据库用户。 角色可以拥有数据库对象(比如表)以及可以把这些对象上的权限赋予其它角色， 以控制谁拥有访问哪些对象的权限。另外，我们也可以把一个角色的成员 权限赋予其它角色，这样就允许成员角色使用分配给另一个角色的权限。

角色的概念替换了"用户"和"组"。在PostgreSQL 版本 8.1 之前，用户和组是独立类型的记录，但现在它们只是角色。 任何角色都可以是一个用户、一个组、或者两者。

本章描述如何创建和管理角色。更多各种数据库对象角色权限效果的信息可以在Section 5.6找到。

20.1. 数据库角色

数据库角色从概念上与操作系统用户是完全无关的。在实际使用中把它们对应起来可能比较方便， 但这不是必须的。数据库角色在整个数据库集群中是全局的(而不是每个库不同)。 要创建一个角色，使用 SQL 命令CREATE ROLE执行：

```
CREATE ROLE _name_;
```

`_name_` 遵循 SQL 标识的规则：要么完全没有特殊字符， 要么用双引号包围(实际上你通常会给命令增加额外的选项，比如 `LOGIN` 。 下面显示更多细节)。要删除一个现有角色，使用类似的DROP ROLE命令：

```
DROP ROLE _name_;
```

为了方便，程序createuser和dropuser 提供了对了这些 SQL 命令的封装。我们可以在 shell 命令上直接调用它们：

```
createuser _name_
dropuser _name_
```

要检查现有角色的集合，可以检查 `pg_roles` 系统表，比如：

```
SELECT rolname FROM pg_roles;
```

psql的元命令 `\du` 也可以用于列出现有角色。

为了能创建初始数据库系统，新建立的数据库总是包含一个预定义的"超级用户"角色， 并且缺省时(除非在运行 `initdb` 时更改过) 他将和初始化该数据库集群的用户有相同的名称。通常，这个角色名叫 `postgres` 。 为了创建更多角色，你必须首先以这个初始用户角色连接。

每一个和数据库的连接都必须用一个角色身份进行，这个角色决定在该连接上的初始权限。特定数据库连接的角色名是在初始化连接请求的时候声明的。比如， `psql` 程序使用 `-U` 命令行选项声明它代表的角色。 许多应用以当前操作系统的用户名为缺省角色名(这样的应用包括 `createuser` 和 `psql`)。 所以，在系统用户和数据库角色之间有某种命名关系会让我们工作方便很多。

一个客户端连接可以使用的角色集合是由客户认证设置决定的， 在Chapter 19里面有解释。因此， 一个客户端并不局限于以它的操作系统用户匹配的角色进行连接， 就像你登录系统的名称不一定要是你的真实姓名一样。因为角色的身份决定了一个连接的权限， 所以在多用户环境里仔细配置这些内容是非常重要的。

20.2. 角色属性

一个数据库角色可以有一系列属性，这些属性定义他的权限，以及与客户认证系统的交互。

登陆权限

只有具有 `LOGIN` 属性的角色才可以用作数据库连接的初始角色名。 一个带有 `LOGIN` 属性的角色可以认为是和"数据库用户"相同的事物。 要创建一个具有登录权限的角色，用下列之一：

```
CREATE ROLE _name_ LOGIN;
CREATE USER _name_;
```

除了 `CREATE USER` 默认赋予 `LOGIN` 之外， `CREATE USER` 等价于 `CREATE ROLE` (默认不赋予 `CREATE ROLE`)。

超级用户状态

数据库超级用户超越所有权限检查。这是一个危险的权限，应该小心使用；最好使用非超级用户完成你的大多数工作。要创建数据库超级用户， 用 `CREATE ROLE` `_name_` SUPERUSER 命令。 你必须用已经是超级用户的角色执行这条命令。

创建数据库

角色要想创建数据库，必须明确给出权限(对于超级用户是例外，因为他们超越所有权限检查)。 要创建这样的角色，用 `CREATE ROLE` `_name_` CREATEDB命令。

创建角色

角色要想创建角色，必须明确给出权限(对于超级用户是例外，因为他们超越所有权限检查)。要创建这样的角色，用 `CREATE ROLE` `_name_` CREATEROLE 命令。一个带有 `CREATEROLE` 权限的角色也可以更改和删除其它角色， 以及给其它角色赋予或者撤销成员关系。不过，要创建、更改、删除一个超级用户角色的成员关系， 需要具有超级用户属性；只有 `CREATEROLE` 还不够。

启动复制

角色要想启动流复制，必须明确给出权限(对于超级用户是例外，因为他们超越所有权限检查)。 用于流复制的角色必须总是拥有 `LOGIN` 权限。要创建这样的角色， 使用 `CREATE ROLE` `_name_` REPLICATION LOGIN命令。

口令

只有在客户认证方法要求与数据库建立连接必须使用口令的时候，口令才比较重要。 `password` 和 `md5` 认证方法使用口令。数据库口令与操作系统口令是无关的。 在创建角色的时候可以这样声明一个口令： `CREATE ROLE` `_name_` PASSWORD ' `_string_` '。

一个角色的属性可以在创建后用 `ALTER ROLE` 修改。参考CREATE ROLE 和ALTER ROLE的手册获取细节。

> **Tip:** 创建一个具有 `CREATEDB` 和 `CREATEROLE` 权限，但是并非超级用户的角色是一个很好的习惯，你可以使用这个角色进行所有日常的数据库和角色管理。这个方法避免了以超级用户操作时，发生误操作导致的严重后果。

一个角色也可以为许多运行时配置设置针对其个人的缺省，那些配置在Chapter 18里描述。比如，如果出于某种原因你想在所有你做的连接中关闭索引扫描(不是个好主意)，你可以用：

```
ALTER ROLE myname SET enable_indexscan TO off;
```

这样就会保存该设置(但是不是立即设置)。然后，在这个角色随后的连接中就好像在会话开始之后都立即 `SET enable_indexscan TO off` 了一样。你也可以在会话中修改这个设置；它只是缺省。要撤销任何这样的设置，使用 `ALTER ROLE` `_rolename_` RESET `_varname_`。请注意，对那些没有 `LOGIN` 属性的角色，这些角色相关的缺省值几乎没什么用，因为它们从来不会被调用。

20.3. 角色成员

把用户组合起来简化权限管理是个常用的便利方法：用这样的方法，权限可以赋予整个组，也可以对整个组撤消。在PostgreSQL里， 这些事情是通过创建代表一个组的角色，然后赋予组角色的成员 权限给独立的用户角色的方法实现的。

要设置一个组角色，首先创建角色：

```
CREATE ROLE _name_;
```

一般作为组使用的角色不应当具有 `LOGIN` 属性， 虽然你可以设置它。

一旦组角色已经存在了，那么你就可以用GRANT 和REVOKE命令添加和撤消权限：

```
GRANT _group_role_ TO _role1_, ... ;
REVOKE _group_role_ FROM _role1_, ... ;
```

你还可以赋予成员权限给其它组角色(因为在组角色和非组角色之间没有实质的区别)。 唯一的制约是你不能建立循环的成员关系。另外，不允许给 `PUBLIC` 角色赋予成员权限。

一个组角色的成员可以用两种方法使用角色的权限。首先， 一个组的每个成员都可以明确用SET ROLE临时"变成"该组的成员。 在这个状态下，数据库会话具有该组角色的权限，而不是原始的登录角色权限， 这个时候创建的数据库对象被认为是由组角色拥有，而不是登录角色。第二， 拥有 `INHERIT` 属性的角色成员自动具有它们所属组角色的权限， 包括任何通过那些角色继承的权限。例如，假如我们做了下面的事情：

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

那么在以角色 `joe` 连接之后，该数据库会话将立即拥有直接赋予 `joe` 的权限加上任何赋予 `admin` 的权限，因为 `joe` "继承" 了 `admin` 的权限。不过，赋予 `wheel` 的权限不可用， 因为即使 `joe` 是 `wheel` 的一个间接成员， 但该成员关系是通过 `admin` 过来的，而该组有 `NOINHERIT` 属性。在：

```
SET ROLE admin;
```

之后，该会话将只拥有那些已赋予 `admin` 的权限， 而不包括那些已赋予 `joe` 的权限。在

```
SET ROLE wheel;
```

之后，该会话将只能使用已赋予 `wheel` 的权限，而不包括已赋予 `joe` 或 `admin` 的权限。原来的权限可以用下列之一恢复：

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> **Note:** `SET ROLE` 命令总是允许选取任意登录角色直接或者间接所在的组角色。 因此，在上面的例子里，我们没必要在变成 `wheel` 之前先变成 `admin` 。
>
> **Note:** 在 SQL 标准里，在用户和角色之间有明确的区别，并且用户并不会自动继承权限，而角色可以。 这个行为在PostgreSQL里面可以通过给予那些当作 SQL 角色使用的角色以 `INHERIT` 属性，而给予当作 SQL 用户使用的角色以 `NOINHERIT` 属性来实现。 不过，PostgreSQL缺省是给予所有角色 `INHERIT` 属性， 目的是和 8.1 之前的版本向下兼容，那些版本里，用户总是能使用他们所在组被赋予的权限。

角色属性 `LOGIN` ， `SUPERUSER` ， `CREATEDB` ， `CREATEROLE` 可以被认为是特殊的权限，但是它们从来不会像数据库对象上的普通权限那样继承。 你必须明确地 `SET ROLE` 到一个特殊的角色，这个角色应该是拥有这些属性的角色， 然后才能利用这些属性。继续上面的例子，我们也可以选择给 `admin` 角色赋予 `CREATEDB` 和 `CREATEROLE` 权限。然后，以 `joe` 连接的会话不会立即有这些权限，只有在 `SET ROLE admin` 之后才有。

要删除一个组角色，用DROP ROLE命令：

```
DROP ROLE _name_;
```

任何在组角色里面的成员关系都会自动撤消(但是成员角色自己则不受影响)。 不过，请注意任何组角色拥有的对象都必须首先删除或者赋予其它所有者；并且任何给该组角色赋予的权限都必须撤消。

20.4. 函数和触发器安全

函数和触发器允许用户向后端服务器插入代码，这样其它用户可以在无意识的情况下执行这些代码。 因此，两种机制都可以让用户相当隐蔽地给别人设置"木马"， 唯一的有效防护就是严格控制谁可以定义函数。

后端服务器里面运行的函数都是以数据库服务器守护进程的操作系统权限运行的。 如果所使用的编程语言允许无检查的内存访问，那么修改服务器的内部数据结构也是可能的。 因此，除了其它问题外，这样的函数可以绕过任何系统访问控制。 允许这样访问的函数语言都被认为是"不可信的"，PostgreSQL 只允许超级用户使用这样的语言书写函数。

Chapter 21. 管理数据库

Table of Contents

每个正在运行的PostgreSQL服务器实例都管理着一个或多个数据库。 因此，在组织SQL对象("数据库对象")的层次中，数据库位于最顶层。 本章描述数据库的属性，以及如何创建、管理、删除它们。

21.1. 概述

数据库是一些SQL对象("数据库对象")的集合； 通常每个数据库对象(表、函数等)属于并且只属于一个数据库。不过有几个系统表 (比如 `pg_database`)属于整个集群并且可以在集群之内的每个数据库里访问。 更准确地说，一个数据库是一个模式的集合，而模式包含表、函数等等。 因此完整的层次是这样的：服务器→数据库→模式→表(或者其它类型对象，比如函数)。

在与数据库服务器连接的时候，应用应该在它的连接请求里指明它想连接的数据库名称。 不允许在一次连接里访问多个数据库(不过没有限制一个应用可以建立的连接数量)。 数据库是物理上相互隔离的，对它们的访问控制是在连接层次进行的。如果一个 PostgreSQL服务器实例用于承载那些应该分隔并且相互之间并不知晓的用户和项目， 那么我们建议把它们放在不同的数据库里。如果项目或者用户是相互关联的， 并且可以相互使用对方的资源，那么应该把它们放在同一个数据库里，但可能在不同的模式中。 模式只是一个纯粹的逻辑结构，谁能访问某个模式由权限系统控制。 有关管理模式的更多信息在Section 5.7里。

数据库是使用 `CREATE DATABASE` 命令创建的(参阅Section 21.2)， 用 `DROP DATABASE` 命令删除(参阅Section 21.5)。 要查看现有数据库的集合，可以检查系统表 `pg_database` ，比如

```
SELECT datname FROM pg_database;
```

psql程序的 `\l` 元命令和 `-l` 命令行选项也可以用来列出现存的数据库。

> **Note:** SQL标准把数据库称作"目录"(catalog)，不过这两个东西实际上没有什么区别。

21.2. 创建一个数据库

为了创建数据库，必须先运行PostgreSQL服务器(参阅Section 17.3)。

数据库是用 SQL 命令CREATE DATABASE创建的：

```
CREATE DATABASE _name_;
```

这里的 `_name_` 遵循SQL标识符的一般规则。 当前角色自动成为此新数据库的所有者。同时，以后删除这个数据库也是这个用户的特权 (同时还会删除其中的所有对象，即使那些对象有不同的所有者也这样)。

创建数据库是一个有限制的操作。参阅Section 20.2获取如何赋予权限的信息。

因为你需要与数据库服务器连接才能执行 `CREATE DATABASE` 命令， 那么还有一个问题是第一个数据库是怎样创建的? 第一个数据库总是由 `initdb` 命令在初始化数据存储区的时候创建的 (参阅Section 17.2)。这个数据库叫 `postgres` 。因此要创建第一个"真正" 的数据库时你可以与 `postgres` 连接。

在数据库集群初始化时会创建另一个名为 `template1` 的数据库。在创建一个新的数据库时，实际上就是克隆(复制)了 `template1` 数据库。 这就意味着你对 `template1` 所做的任何修改都会传播到所有随后创建的数据库中。 因此，避免在 `template1` 中创建对象，除非你想要这些对象传播到每个新建的数据库中。 更多细节见Section 21.3。

另外，为了方便，你还可以在shell中用 `createdb` 程序来创建新数据库：

```
createdb _dbname_
```

`createdb` 没变什么魔术，它和 `postgres` 连接并执行 `CREATE DATABASE` 命令，就像上面描述的那样。createdb的手册页包含使用它的细节。 注意不带任何参数调用 `createdb` 将创建与当前用户名同名的数据库。

> **Note:** Chapter 19包含有关如何限制哪些用户可以连接某个特定数据库的信息。

有时候你想为其它人创建一个数据库，并且使他应该成为新数据库的所有者， 这样他就可以自己配置和管理这个数据库。要实现这个目标，使用下列命令中的一条：

```
CREATE DATABASE _dbname_ OWNER _rolename_;
```

用于 SQL 环境，或：

```
createdb -O _rolename_ _dbname_
```

用于命令行。只有数据库的超级用户才能为其它用户创建数据库。

21.3. 模板数据库

`CREATE DATABASE` 实际上是通过拷贝一个现有的数据库进行工作的。缺省时， 它拷贝名为 `template1` 的标准系统数据库。 所以该数据库是创建新数据库的"模板"。如果你给 `template1` 增加对象， 这些对象将被拷贝到随后创建的用户数据库中。这样的行为允许节点对数据库中的标准套件进行修改。 比如，如果你把过程语言PL/Perl安装到 `template1` 里，那么你在创建用户数据库的时候它们就会自动可得，而不需要额外的动作。

系统里还有名为 `template0` 的第二个标准系统数据库， 这个数据库包含和 `template1` 初始时一样的数据内容，也就是说， 只包含标准的PostgreSQL对象。在数据库集群初始化之后，我们不应该对 `template0` 做任何修改。通过告诉 `CREATE DATABASE` 使用 `template0` 而不是 `template1` 进行拷贝，你可以创建一个"纯净" 的用户数据库，它不会包含任何 `template1` 里所特有的东西。 这一点在恢复 `pg_dump` 转储的时候是非常方便的： 转储脚本应该在一个纯净的数据库中恢复以确保我们正确创建了被转储出的数据库内容， 而不会随后和可能已经添加到 `template1` 中的对象相冲突。

拷贝 `template0` 而不是 `template1` 的另一个常见原因是在拷贝 `template0` 时，可以指定新的编码和本地设置，而拷贝 `template1` 时必须使用和 `template1` 相同的设置。这是因为 `template1` 可能包含编码指定或本地指定的数据，而 `template0` 不这样。

要通过拷贝 `template0` 的方法创建一个数据库，可使用：

```
CREATE DATABASE _dbname_ TEMPLATE template0;
```

用于 SQL 环境，或：

```
createdb -T template0 _dbname_
```

用于 shell 环境。

我们可以创建额外的模板数据库，而且实际上我们可以在一个集群中通过将 `CREATE DATABASE` 的模板声明为相应的数据库名拷贝任何数据库。不过， 我们必需明白，这个功能并非一般性的" `COPY DATABASE` "工具。 实际上，在拷贝操作的过程中，源数据库必需是空闲状态(没有正在处理的数据修改事务)。 如果在 `CREATE DATABASE` 开始的时候存在其它连接，那么操作将会失败， 在拷贝期间，到源数据库的新连接都被阻止。

在 `pg_database` 里有两个有用的标志可以用于每个数据库： `datistemplate` 和 `datallowconn` 字段。 `datistemplate` 表示该数据库是准备用作 `CREATE DATABASE` 模板的。如果设置了这个标志， 那么该数据库可以由任何有 `CREATEDB` 权限的用户克隆；如果没有设置， 那么只有超级用户和该数据库的所有者可以克隆它。如

果`datallowconn`为假， 那么将不允许与该数据库发生任何新的连接(不过现有的会话不会因为把该标志设置为假而终止)。 `template0`数据库通常被标记为`datallowconn = false`以避免对它的修改。 `template0`和`template1`都应该总是标记为`datistemplate = true`。

> **Note:** `template1`和`template0`没有任何特殊的状态， 除了`template1`是`CREATE DATABASE`的缺省源数据库名之外。比如， 我们可以删除`template1`然后从`template0`中创建它而不会有任何不良效果。 如果我们不小心在`template1`里加了一堆垃圾，那么我们就会建议做这样的操作。 （要删除`template1`，必须使`pg_database.datistemplate = false`。）
>
> 在初始化数据库集群的时候，也会创建`postgres`数据库。 这个数据库用于做为用户和应用连接的缺省数据库。它只是`template1`的一个简单拷贝， 需要的时候可以删除或者重建。

21.4. 数据库配置

回顾一下Chapter 18 我们知道PostgreSQL 服务器提供了大量的运行时配置变量。你可以为许多这样的变量设置特定于数据库的缺省数值。

比如，如果由于某种原因，你想关闭某个数据库上的GEQO优化器， 你就不得不要么在一开始就在所有数据库中关闭它， 要么是保证每个连接过来的客户端都很小心地发出了 `SET geqo TO off` 命令。 要令这个设置在特定数据库里成为缺省，你可以执行下面的命令：

```
ALTER DATABASE mydb SET geqo TO off;
```

这样将保存该设置(但不是立即设置它)。在随后的连接中它将表现的像在会话开始后马上调用 `SET geqo TO off;` 一样。请注意用户仍然可以在该会话中更改这个设置(它只是缺省)。 要撤消这样的设置，使用 `ALTER DATABASE` `_dbname_` RESET `_varname_` 。

21.5. 删除数据库

数据库是用DROP DATABASE命令删除的：

```
DROP DATABASE _name_;
```

只有数据库的所有者或者超级用户才可以删除数据库。 删除数据库会删除数据库中包括的所有对象。数据库的删除是不可恢复的。

你不能在与目标库连接的时候执行 `DROP DATABASE` 命令。不过， 你可以和其它数据库连接，包括 `template1` 数据库。 `template1` 也是你删除集群中最后一个库的唯一方法。

为了方便，有一个在 shell 上运行的删除数据库的dropdb程序：

```
dropdb _dbname_
```

它和 `createdb` 不一样，没有缺省删除的数据库名称。

21.6. 表空间

PostgreSQL里的表空间允许数据库管理员在文件系统里定义那些代表数据库对象的文件存放位置。 一旦创建了表空间，那么就可以在创建数据库对象的时候引用它。

通过使用表空间，管理员可以控制一个PostgreSQL安装的磁盘布局。 这么做至少有两个用处。首先，如果初始化集群所在的分区或者卷用光了空间， 而又不能扩展，那么表空间可以在一个不同的分区上创建和使用， 直到系统可以重新配置。

第二，表空间允许管理员根据数据库对象的使用模式安排数据位置，从而优化性能。比如，一个很频繁使用的索引可以放在非常快并且非常可靠的磁盘上，比如一种非常贵的固态设备。而同时， 一个存储归档的数据，很少使用的或者对性能要求不高的表可以存储在一个便宜但比较慢的磁盘系统上。

要定义一个表空间，使用CREATE TABLESPACE命令，比如：

```
CREATE TABLESPACE fastspace LOCATION '/mnt/sda1/postgresql/data';
```

这个位置必须是一个现有的空目录，并且属于PostgreSQL系统用户。 所有随后在该表空间创建的对象都将被存放在这个目录下的文件里。

> **Note:** 通常在一个逻辑文件系统上建立多个表空间没有什么意义， 因为无法控制一个逻辑文件系统里不同文件的位置。不过，PostgreSQL 并不做这方面的任何强制，并且它实际上并不知道文件系统边界。 它只知道在指定的目录里存储文件。

创建表空间本身必须用数据库超级用户身份进行，但之后你就可以允许普通数据库用户利用它了。 要做这件事情，在表空间上给这些用户授予 `CREATE` 权限。

表、索引和整个数据库都可以放在特定的表空间里。想要这么做的话， 在给定表空间上有 `CREATE` 权限的用户必须把表空间的名字以一个参数的形式传递给相关的命令。 比如，下面的命令在表空间 `space1` 上创建一个表：

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

另外，还可以使用default_tablespace参数：

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

只要 `default_tablespace` 被设置为非空字符串， 那么它就为没有明确使用 `TABLESPACE` 子句的 `CREATE TABLE` 和 `CREATE INDEX` 命令提供一个隐含的子句。

也有一个temp_tablespaces参数，决定临时表和索引的放置， 就和用于像存储大数据集这样的目的临时文件一样。这可能是一个表空间名字的列表， 而不是只有一个名字，所以与临时对象相关联的负载可以散布到多个表空间中。 每次创建临时对象时选择一个随机的成员列表。

与一个数据库相关联的表空间用于存储该数据库的系统表。另外， 如果没有给出 `TABLESPACE` 子句，并且没有通过 `default_tablespace` 或 `temp_tablespaces` （视情况而定）指定其他选项，那么在数据库中创建表， 索引和临时文件时使用的是缺省表空间。如果创建数据库时没有给它声明一个表空间， 那么它使用与它拷贝的模版数据库相同的表空间。

当数据库集群初始化时，自动创建两个表空间。 `pg_global` 表空间用于共享的系统表。 `pg_default` 是 `template1` 和 `template0` 数据库的缺省表空间 (因此，这个表空间也将是任何其它数据库的缺省表空间，除非在 `CREATE DATABASE` 中通过 `TABLESPACE` 子句重写)。

创建了表空间之后，它就可以用于任何数据库，只要请求的用户有足够权限。 这意味着除非我们把使用这个表空间的所有数据库里的所有对象都删除掉，否则我们不能删除该表空间。

要删除一个空的表空间，使用DROP TABLESPACE命令。

检查 `pg_tablespace` 系统表就可以获取现有的表空间，比如

```
SELECT spcname FROM pg_tablespace;
```

psql程序的 `\db` 元命令也可以用于列出现有表空间。

为了简化表空间的实现，PostgreSQL使用了符号连接。 这就意味着表空间只能在支持符号连接的系统上使用。

目录 `$PGDATA/pg_tblspc` 包含指向集群里定义的每个非内置表空间的符号连接。 尽管我们不建议，但是我们还是可能通过手工重定义这些连接来调整表空间的布局。 在服务器运行的时候不要这么干。注意在PostgreSQL 9.1及以前，你仍需要用新的位置更新 `pg_tablespace` 表。如果你不这么做， `pg_dump` 将继续显示旧的表空间位置。

Chapter 22. 区域

Table of Contents

本章从管理员的角度描述可用的区域特性。PostgreSQL 支持2种区域设置：

- 利用操作系统的区域(locale)特性，提供区域特定的排序顺序、数值格式、本地消息和其它方面的支持。 这些都位于Section 22.1和Section 22.2中。
- 提供多种不同的字符集以支持存储各种语言的文本、 并且提供客户端和服务器端之间的字符集转换。 这些位于Section 22.3中。

22.1. 区域支持

区域支持指的是应用中考虑字母、排序、数值格式化等与文化相关的问题。 PostgreSQL使用服务器操作系统提供的标准ISO C 和POSIX区域机制。 更多的信息请参考你的系统文档。

22.1.1. 概述

区域支持是在使用 `initdb` 创建一个数据库集群的时候自动初始化的。 缺省时， `initdb` 将会按照它的执行环境的区域设置初始化数据库集群；因此如果你的系统已经设置为你的数据库集群想要的区域，那么你就没有什么可干的了。 如果你想使用其它的区域(或者你还不知道你的系统设置的区域是什么)， 那么你可以用 `--locale` 命令行选项告诉 `initdb` 你需要的区域究竟是哪个。 比如：

```
initdb --locale=sv_SE
```

Unix系统下的这个例子就把区域设置为瑞典语(`sv`)，并且在瑞典地区(`SE`)。 其它的可能性是 `en_US` (美国英语)和 `fr_CA` (加拿大法语)等等。 如果有多于一种的字符集可以用于同一个区域，那么声明看起来会像 `_language_territory.codeset_` 。 比如， `fr_BE.UTF-8` 表示为在比利时(BE)地区使用的法语(fr)，并且使用UTF-8字符集编码。

你的系统里有哪些可用的区域设置，它们的名字是什么， 这些信息都取决于你的操作系统提供商提供了什么以及你安装了什么东西。 在大多数系统上，命令 `locale -a` 将提供所有可用区域的一个列表。 Windows使用更详细的区域名称，比如 `German_Germany` 或者 `Swedish_Sweden.1252` ， 但是原则是一样的。

有时候，把几种区域规则混合起来也很有用，比如， 使用英语排序规则而用西班牙语消息。为了支持这些， 我们有一套区域子范畴用于控制区域规则的某一方面：

`LC_COLLATE`	字符串排序顺序
`LC_CTYPE`	字符分类(什么是字母?是否区分大小写?)
`LC_MESSAGES`	消息的语言
`LC_MONETARY`	货币金额的格式
`LC_NUMERIC`	数值格式
`LC_TIME`	日期和时间格式

这些范畴名转换成 `initdb` 选项的名字以覆盖某个特定范畴的区域选择。 比如，要把区域设置为加拿大法语，但使用美国的货币格式化规则， 可以使用 `initdb --locale=fr_CA --lc-monetary=en_US` 。

如果你想要你的系统表现得像没有区域支持一样，那么使用特殊的区域 `C` 或 `POSIX` 。

一些区域范畴的值必须在创建数据库时固定下来。 您可以对不同的数据库使用不同的设置，但一旦创建一个数据库，你就再也不能更改它们了。 `LC_COLLATE` 和 `LC_CTYPE` 就是这样的范畴。 它们影响索引的排序顺序，因此它们必需保持固定，否则在文本字段上的索引将会崩溃。 （但是你可以使用排序规则(collation)缓解这种限制，正如Section 22.2中讨论的）。 当运行 `initdb` 时确定这些范畴的缺省值， 这些值被用于创建新的数据库，除非在 `CREATE DATABASE` 命令中明确指定。

其它区域范畴可以在服务器启动的时候根据需要设置 服务器配置参数来改变(参阅Section 18.11.2获取细节)。 `initdb` 选择的值实际上只是作为服务器启动时的缺省值写入 `postgresql.conf` 配置文件。 如果你在 `postgresql.conf` 里面删除了这些设置， 那么服务器将会继承来自运行环境的设置。

请注意服务器的区域行为是由它看到的环境变量决定的， 而不受客户端的环境影响。 因此，我们要在启动服务器之前认真地设置好这些变量。 这样带来的一种情况是如果客户端和服务器设置成不同的区域， 那么消息可能以不同的语言呈现，这取决于消息的来源。

> **Note:** 在我们谈到从执行环境继承区域的时候， 我们的意思是在大多数操作系统上的下列动作： 对于一个给定的区域范畴， 比如排序规则，按照下面的顺序评估这些环境变量， 直到找到一个已设置的： `LC_ALL` ， `LC_COLLATE` (或者对应于相应范畴的其他变量)， `LANG` 。 如果这些环境变量一个都没有设置，那么区域缺省为 `C` 。
>
> 一些消息本地化库也使用环境变量 `LANGUAGE` ， 它覆盖所有其它用于设置语言信息的区域设置。 如果有问题，请参考你的操作系统文档， 特别是gettext的文档以获取更多信息。

要能够将消息翻译成用户选择的语言，编译时必需选择NLS选项(`configure --enable-nls`)。其它区域支持是自动包含的。

22.1.2. 行为

区域设置特别影响下面的 SQL 特性：

- 查询中使用 `ORDER BY` 或者对文本数据的标准比较操作符进行排序
- `upper` ， `lower` 和 `initcap` 函数
- 模式匹配运算符(`LIKE` ， `SIMILAR TO` ，以及POSIX-风格的正则表达式)； 区域影响大小写不敏感的匹配和通过字符分类正则表达式的字符分类。
- `to_char` 函数族
- 使用 `LIKE` 子句的索引能力

PostgreSQL里使用非 `C` 或者 `POSIX` 区域的缺点是性能影响。 它降低了字符处理的速度并阻止了在 `LIKE` 类查询里面普通索引的使用。 因此，应该只有在你实际上需要的时候才使用它。

为了允许PostgreSQL在非C区域下的 `LIKE` 子句中使用索引， 有好几个自定义的操作符类可以用。 这些操作符类允许创建一个严格地比较每个字符的索引， 而忽略区域比较规则。请参考Section 11.9获取更多信息。 另外一个方法是使用 `C` collation创建索引，正如Section 22.2中讨论的。

22.1.3. 问题

如果经过上面解释后区域支持仍然不能运转， 那你就要检查一下操作系统的区域支持是否正确配置。 要检查某个区域是否安装并且正常运转， 你可以使用 `locale -a` 命令(如果你的系统提供了该命令)。

请检查核实PostgreSQL确实使用了你认为它该用的区域设置。 `LC_COLLATE` 和 `LC_CTYPE` 的设置都是在数据库创建时决定的， 不能被改变除非创建新的数据库。其它的区域设置包括 `LC_MESSAGES` 和 `LC_MONETARY` 都是由服务器的启动环境决定的， 但是可以在运行时修改。你可以用 `SHOW` 命令检查数据库正在使用的区域设置。

源码发布中的 `src/test/locale` 目录包含 PostgreSQL的区域支持测试套件。

那些通过解析错误消息文本处理服务器端错误的客户端应用很明显会有问题， 因为服务器信息可能会以不同的语言表示。我们建议这类应用的开发人员改用错误代码机制。

维护消息翻译表需要许多志愿者的坚持不懈的努力， 他们就是希望PostgreSQL以他们的语言说话的人。 如果你的语言消息目前还不可用或者没有完全翻译完成， 那么我们很欢迎你的协助。如果你想帮忙， 那么请参考Chapter 50或者向开发者邮递列表发邮件。

22.2. 排序规则支持

排序规则特性允许为每一列数据指定排序顺序和字符分类行为，或者甚至为每个操作指定。这缓解了 `LC_COLLATE` 和 `LC_CTYPE` 在数据库被创建后不能被修改的限制。

22.2.1. 概念

从概念上讲，collatable数据类型的每个表达式都有一个排序规则。（内置collatable数据类型有 `text`，`varchar` 以及 `char`。 用户定义的基本类型，也可以标记为collatable， 当然一个collatable数据类型的域也是collatable的）。 如果表达式是一个列引用，该表达式的排序规则是就这个列的排序规则。 如果表达式是一个常数，排序规则是常数数据类型的缺省排序规则。 一个更复杂表达式的排序规则从它的输入端排序规则推导，如下所述。

一个表达式的排序规则可以是"缺省"排序规则， 这意味着为数据库的区域设置。它也可以用于表达式的排序规则是不确定的。 在这种情况下，排序操作符以及其他需要知道排序规则的操作符会在执行时失败。

当数据库系统必须执行排序或字符分类时，它使用输入表达式的排序规则。 这种情况发生，例如，使用 `ORDER BY` 子句 以及函数或运算符调用比如 `<`。 应用到 `ORDER BY` 子句中的排序规则直接就是排序关键字的排序规则。 应用到函数或操作符调用的排序规则要从参数上派生， 如下文所述。除了比较操作符外， 大小写字母转换函数，如 `lower`，`upper` 和 `initcap`；模式匹配运算符号；以及 `to_char` 及其相关函数都需要考虑排序规则。

对于一个函数或运算符调用， 通过检查用来在运行时执行指定操作的参数排序规则派生出该排序规则。 如果函数或运算符调用结果是collatable数据类型， 并且有需要知道该排序规则的外围表达式， 排序规则也可用于在解析时作为函数或运算符表达式的定义的排序规则。

表达式的排序规则推导可以是 隐式或显式的。当多个不同的排序规则出现在表达式中，这种区别会影响排序规则如何组合， 当使用 `COLLATE` 子句的时候，产生显式排序规则推导;所有其他排序规则 推导是隐式的。当多个排序规则必须结合时，例如在一个函数调用中，使用下面的规则：

1. 如果任何输入表达式有显式的排序规则推导，那么 输入表达式中所有显式派生的排序规则必须是一样的， 否则将引发错误。如果有任何显式派生的排序规则，这即是排序规则组合的结果。

2. 否则，所有输入表达式必须具有相同的隐式排序规则推导或者缺省排序规则。如果出现任何非缺省的 排序规则，则是这就是排序组合的结果。否则，结果是缺省排序规则。

3. 如果在输入的表达式之间非缺省的隐式排序规则有冲突， 那么该组合视为不明确的排序规则。 这不是一个错误情况，除非被调用的那个函数需要知道排序规则。 如果是这样，在运行时将引发一个错误。

比如，考虑这个表定义：

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

然后

```
SELECT a < 'foo' FROM test1;
```

按照 `de_DE` 规则，执行 < 比较，因为表达式 组合隐式推导排序规则与缺省排序规则，但是

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

使用 `fr_FR` 规则执行比较，因为显式排序规则覆盖了隐式的。 此外，

```
SELECT a < b FROM test1;
```

解析器无法确定应该应用哪个排序规则，因为 `a` 列和 `b` 列拥有冲突的隐式排序规则。 因为 < 操作符确实需要知道所使用的排序规则，这将导致一个错误。 错误可以通过附加一个明确的排序规则说明符给输入表达式得以解决，如下：

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

或者等效的，

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

另一方面，结构上类似的情况

```
SELECT a || b FROM test1;
```

不会产生错误，因为 `||` 操作符并不关心排序规则： 不论什么排序规则结果都是一样的。

如果函数或者操作符的结果还是collatable数据类型, 那么分配给函数或者操作符的组合的输入表达式的排序规则也会应用到函数或者操作符结果上。 因此，

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

按照 `de_DE` 规则执行该排序。但是这个查询：

```
SELECT * FROM test1 ORDER BY a || b;
```

导致一个错误，因为即使 `||` 操作符不需要知道排序规则，而 `ORDER BY` 子句 确实需要。和前面一样，可以使用显式的排序规则说明符解决这个冲突。

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

22.2.2. 管理排序规则

排序规则是把SQL名称映射到操作系统区域的SQL模式对象。 特别是，它映射到 `LC_COLLATE` 和 `LC_CTYPE` 的组合。 （顾名思义，排序规则的主要目的是设置 `LC_COLLATE`，它控制排序顺序。但在实际中很少需要一个和 `LC_COLLATE` 不同的 `LC_CTYPE` 设置， 所以将它们合在一起，而不是为每个表达式另外创造一个设置 `LC_CTYPE` 的基础设施，更为方便）。 另外，排序规则和字符集编码（参见Section 22.3） 紧密关联。 对不同的编码可能存在同名的排序规则。

在所有平台上，名称为 `default`，`C` 和 `POSIX` 排序规则都是可用的。其他可用的排序规则 取决于操作系统支持。 `default` 排序规则选择在创建数据库时指定的 `LC_COLLATE` 和 `LC_CTYPE` 值。 `C` 和 `POSIX` 的排序规则都表现为"传统 C"的行为， 即只有ASCII字母" `A` "到" `Z` " 被视为字母，并且严格按照字符的编码字节值进行排序。

如果操作系统提供了在一个程序中使用多语言环境的支持 (`newlocale` 以及相关函数)，那么当一个数据库集群初始化的时候， `initdb` 使用基于当时在操作系统上发现的所有语言环境的排序规则 来填充系统表 `pg_collation` 。 例如，操作系统可能提供名为 `de_DE.utf8` 的区域。 那么， `initdb` 就可能为编码 `UTF8` 创建命名为 `de_DE.utf8` 的排序规则，它的 `LC_COLLATE` 和 `LC_CTYPE` 都被设置为 `de_DE.utf8` 。 它还将创建名称中被剥离了 `.utf8` 标签的排序规则。 所以，你也可以使用 `de_DE` 名称的排序规则 ，这方便于编写并且使得名称较少依赖于编码。然而，需要注意的是， 排序规则名称的初始设置是平台相关的。

在需要有不同的 `LC_COLLATE` 和 `LC_CTYPE` 值的排序规则的情况下，可以使用CREATE COLLATION 命令创建新的排序规则。 该命令也可以从现有排序规则中创建一个新的，这可能是有用的，以便能够 在应用中使用操作系统无关的排序规则名称。

在任何特定的数据库中，只关心使用该数据库编码的排序规则。 `pg_collation` 中的其他项目会被忽略。因此，剥离了编码名的排序规则 名称，如 `de_DE` ，在一个给定的数据库中也算是独一无二的，即使它不是全局唯一的。 建议使用剥离过的排序规则名称，因为如果你以后决

定改变到另一个数据库编码，可以少改变一样东西。 但是请注意，无论是什么数据库编码，都可以使用 `default` ，`C` 和 `POSIX` 排序规则。

PostgreSQL认为不同排序规则 对象是不兼容的，即使他们有相同的属性。 例如：

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

将引起一个错误，即使 `C` 和 `POSIX` 排序规则具有完全相同的行为。因此不推荐混合剥离的和非剥离的排序规则名。

22.3. 字符集支持

PostgreSQL中的字符集支持可以让你以各种字符集存储文本（也称为编码）， 包含单字节字符集，比如ISO-8859系列和多字节字符集比如EUC(扩展Unix编码)、 UTF-8、Mule国际编码。 所有字符集都可以被客户端透明地使用。 但是有一些不支持在服务器上使用（即作为服务器端编码）。 缺省的字符集是在使用 `initdb` 初始化数据库集群的时候选择的。 在你创建数据库的时候是可以覆盖这个缺省值的。因此，你可以有多个数据库， 每个都有不同的字符集。

但是，有一个重要的限制，每个数据库的字符集必须与该数据库的 `LC_CTYPE` （字符类别）以及 `LC_COLLATE` （字符串排序顺序）区域设置相兼容。对于 `C` 或者 `POSIX` 区域，允许任何字符集， 但对于其他区域只有一个字符集设置能正常工作。 （不过在Windows上，UTF-8编码可用于任何区域）。

22.3.1. 支持的字符集

Table 22-1显示了可以在PostgreSQL中使用的 字符集。

Table 22-1. PostgreSQL字符集

名字	描述	语言	服务端?	字节数/字符	别名
`BIG5`	大五码	繁体中文	No	1-2	`WIN950` , `Windows950`
`EUC_CN`	扩展UNIX代码-CN	简体中文	Yes	1-3	
`EUC_JP`	扩展UNIX代码-JP	日文	Yes	1-3	
`EUC_JIS_2004`	扩展UNIX代码-JP, JIS X 0213	日文	Yes	1-3	
`EUC_KR`	扩展UNIX代码-KR	韩文	Yes	1-3	
`EUC_TW`	扩展UNIX代码-TW	繁体中文，台湾	Yes	1-3	
`GB18030`	国标码	中文	No	1-2	
`GBK`	扩展国标码	简体中文	No	1-2	`WIN936` ,

GBK	扩展国标码	简体中文	No	1-2	Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	拉丁/西里尔语	Yes	1	
ISO_8859_6	ISO 8859-6, ECMA 114	拉丁/阿拉伯语	Yes	1	
ISO_8859_7	ISO 8859-7, ECMA 118	拉丁/希腊语	Yes	1	
ISO_8859_8	ISO 8859-8, ECMA 121	拉丁/希伯莱语	Yes	1	
JOHAB	JOHAB	韩语	No	1-3	
KOI8R	KOI8-R	西里尔语(俄国)	Yes	1	KOI8
KOI8U	KOI8-U	西里尔语(乌克兰)	Yes	1	
LATIN1	ISO 8859-1, ECMA 94	西欧语	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	中欧语	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	南欧语	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	北欧语	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	土耳其语	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	日耳曼语	Yes	1	ISO885910
LATIN7	ISO 8859-13	波罗的海语	Yes	1	ISO885913
LATIN8	ISO 8859-14	凯尔特语	Yes	1	ISO885914
LATIN9	ISO 8859-15	带有欧洲语系和语调的LATIN1	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	罗马尼亚语	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	多语种Emacs	Yes	1-4	
SJIS	Shift JIS	日语	No	1-2	Mskanji , ShiftJIS , WIN932 ,

SHIFT_JIS_2004	Shift JIS, JIS X 0213	日语	No	1-2	
SQL_ASCII	unspecified (see text)	任意	Yes	1	
UHC	Unified Hangul Code	韩语	No	1-2	WIN949 , Windows949
UTF8	Unicode, 8-bit	全部	Yes	1-4	Unicode
WIN866	Windows CP866	西里尔语	Yes	1	ALT
WIN874	Windows CP874	泰国语	Yes	1	
WIN1250	Windows CP1250	中欧语	Yes	1	
WIN1251	Windows CP1251	西里尔语	Yes	1	WIN
WIN1252	Windows CP1252	西欧语	Yes	1	
WIN1253	Windows CP1253	希腊语	Yes	1	
WIN1254	Windows CP1254	土耳其语	Yes	1	
WIN1255	Windows CP1255	希伯来语	Yes	1	
WIN1256	Windows CP1256	阿拉伯语	Yes	1	
WIN1257	Windows CP1257	波罗的语	Yes	1	
WIN1258	Windows CP1258	越南语	Yes	1	ABC , TCVN , TCVN5712 , VSCII

并非所有API都支持上面列出的编码。比如， PostgreSQL JDBC驱动就不支持 MULE_INTERNAL ， LATIN6 ， LATIN8 和 LATIN10 。

SQL_ASCII 设置与其它设置表现得相当不同。如果服务器字符集是 SQL_ASCII ， 服务器根据ASCII标准解析0-127的字节值，而字节值为128-255的则当作未解析的字符。 如果设置为 SQL_ASCII 就不会有编码转换。因此，这个设置基本不用来声明所使用的编码， 因为这个声明会忽略编码。在大多数情况下，如果你使用了任何非ASCII数据， 那么使用 SQL_ASCII 设置都是不明智的，因为PostgreSQL 会无法帮助你转换或者校验非ASCII字符。

22.3.2. 设置字符集

`initdb` 为一个PostgreSQL 集群定义缺省的字符集（编码），比如：

```
initdb -E EUC_JP
```

把缺省字符集设置为 `EUC_JP` (用于日文的扩展Unix编码)。 如果你喜欢用长选项声明的话，可以用 `--encoding` 代替 `-E` 选项。 如果没有给出 `-E` 或者 `--encoding` 选项， `initdb` 将基于指定的或者缺省的区域试图判断合适的编码。

你可以在数据库创建时指定非缺省编码，但是指定的编码必须与所选的区域相兼容：

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

将创建一个使用 `EUC_KR` 字符集以及 `ko_KR` 区域的 名字叫 `korean` 的数据库。另外一种实现方法是使用SQL命令：

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.eu
```

注意上述命令声明拷贝 `template0` 数据库。当拷贝任何其他数据库时， 来自源数据库的编码和区域设置不能被改变，因为可能导致数据损坏。参阅Section 21.3 获取更多信息。

数据库的编码是存储在 `pg_database` 系统表中的。 你可以用 `psql` 的 `-l` 选项或 `\l` 命令列出这些编码。

```
$ <kbd class="literal">psql -l</kbd>
                                         List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |          Access Privilege
-----------+----------+-----------+-------------+-------------+--------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc
(7 rows)
```

> **Important:** 在大多数现代操作系统中，PostgreSQL可以通过 `LC_CTYPE` 的设置决定使用哪种字符集，并且强制只使用匹配的数据库编码。在旧的操作系统上 你有责任确保使用所选区域所期望的编码。 如果这上面犯错误很可能导致与区域相关的操作表现出古怪的行为，比如排序。
>
> 即使当 `LC_CTYPE` 不是 `C` 或者 `POSIX` 时， PostgreSQL也允许超级用户创建 使用 `SQL_ASCII` 编码的数据库。正如 以上所述， `SQL_ASCII` 不强制存储在数据库中的数据具有任何特定的编码，所以这个选择带来 区域相关的不当行为的风险。使用这样的设置组合是不推荐的，也许有一天会被完全禁止。

22.3.3. 服务器和客户端之间的自动字符集转换

PostgreSQL支持在服务器和前端之间的自动编码转换。 转换信息在系统表 `pg_conversion` 中存储。 PostgreSQL带着一些预定义的转换。 它们在Table 22-2中列出。 你可以使用SQL命令 `CREATE CONVERSION` 创建一个新的转换。

Table 22-2. 客户/服务器字符集转换

服务器字符集	可用客户端字符集
`BIG5`	不支持做服务器端编码
`EUC_CN`	*EUC_CN*, `MULE_INTERNAL` , `UTF8`
`EUC_JP`	*EUC_JP*, `MULE_INTERNAL` , `SJIS` , `UTF8`
`EUC_KR`	*EUC_KR*, `MULE_INTERNAL` , `UTF8`
`EUC_TW`	*EUC_TW*, `BIG5` , `MULE_INTERNAL` , `UTF8`
`GB18030`	不支持做服务器端编码
`GBK`	不支持做服务器端编码
`ISO_8859_5`	*ISO_8859_5*, `KOI8R` , `MULE_INTERNAL` , `UTF8` , `WIN866` , `WIN1251`
`ISO_8859_6`	*ISO_8859_6*, `UTF8`
`ISO_8859_7`	*ISO_8859_7*, `UTF8`
`ISO_8859_8`	*ISO_8859_8*, `UTF8`
`JOHAB`	*JOHAB*, `UTF8`
`KOI8R`	*KOI8R*, `ISO_8859_5` , `MULE_INTERNAL` , `UTF8` , `WIN866` , `WIN1251`
`KOI8U`	*KOI8U*, `UTF8`
`LATIN1`	*LATIN1*, `MULE_INTERNAL` , `UTF8`
`LATIN2`	*LATIN2*, `MULE_INTERNAL` , `UTF8` , `WIN1250`
`LATIN3`	*LATIN3*, `MULE_INTERNAL` , `UTF8`

LATIN4	*LATIN4*, MULE_INTERNAL , UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5 , EUC_CN , EUC_JP , EUC_KR , EUC_TW , ISO_8859_5 , KOI8R , LATIN1 to LATIN4 , SJIS , WIN866 , WIN1250 , WIN1251
SJIS	不支持做服务器端编码
SQL_ASCII	任意*(不会发生编码转换)*
UHC	不支持做服务器端编码
UTF8	所有支持的编码
WIN866	*WIN866*, ISO_8859_5 , KOI8R , MULE_INTERNAL , UTF8 , WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2 , MULE_INTERNAL , UTF8
WIN1251	*WIN1251*, ISO_8859_5 , KOI8R , MULE_INTERNAL , UTF8 , WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

要想打开自动字符集转换功能，你必须告诉PostgreSQL 你想在客户端使用的字符集(编码)。你可以用好几种方法实现这个目的。

- 用psql里的 `\encoding` 命令。 `\encoding` 允许你动态修改客户端编码。 比如，把编码改变为 `SJIS` ，键入：

```
\encoding SJIS
```

- 使用libpq (Section 31.10)函数控制客户端编码。

- 使用 `SET client_encoding TO` 。使用下面的SQL命令设置客户端编码：

  ```
  SET CLIENT_ENCODING TO '_value_';
  ```

  你也可以使用标准的SQL语法 `SET NAMES` 达到这个目的：

  ```
  SET NAMES '_value_';
  ```

  查询当前客户端编码：

  ```
  SHOW client_encoding;
  ```

  返回缺省编码：

  ```
  RESET client_encoding;
  ```

- 使用 `PGCLIENTENCODING` 。如果在客户端的环境里定义了 `PGCLIENTENCODING` 环境变量，那么在与服务器进行连接时将自动选择这个客户端编码。 这个编码随后可以用上面谈到的任何其它方法覆盖。
- 使用client_encoding配置变量。 如果在 `client_encoding` 里设置了该变量， 那么在与服务器建立了连接之后，将自动选定这个客户端编码。 这个设置随后可以被上面提到的其它方法覆盖。

假如无法进行特定的字符转换— 比如， 你选的服务器编码是 `EUC_JP` 而客户端是 `LATIN1` ，那么有些返回的日文字符不能转换成 `LATIN1` —这时将报告错误。

如果客户端字符集定义成了 `SQL_ASCII` ， 那么编码转换会被关闭，不管服务器的字符集是什么都一样。 和服务器一样，除非你的工作环境全部是ASCII数据，否则使用 `SQL_ASCII` 是不明智的。

22.3.4. 进一步阅读

下面是学习各种类型的编码系统的好地方。

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

包含 `EUC_JP` ， `EUC_CN` ， `EUC_KR` ， `EUC_TW` 的详细说明。

http://www.unicode.org/

Unicode主页。

RFC 3629

UTF-8 (8-bit UCS/Unicode转换格式)的定义。

Chapter 23. 日常数据库维护工作

Table of Contents

像许多其它数据库一样，PostgreSQL也需要周期性的运行某些任务以 实现性能优化。这里讨论的任务是必须经常重复的事情， 可以很容易的使用标准的工具(比如cron脚本)或Windows的任务计划来完成。不过， 设置合适的脚本以及检查它们是否成功执行则是数据库管理员的责任。

一件很明显的维护工作就是经常性地创建数据的备份拷贝。如果没有最近的备份， 那么你就没有从灾难中恢复的机会(磁盘坏、失火、误删表)。可以在PostgreSQL 里面使用的备份和恢复机制在Chapter 24里面有比较详细的讨论。

其它主要的维护工作包括周期性的"vacuuming"(清理)数据库。 这个工作我们在Section 23.1里讨论。与此紧密相关的是更新规划器使用的统计信息， 这个在Section 23.1.3里讨论。

其它需要周期性注意的东西是日志文件的管理。我们在Section 23.3里讨论了这个问题。

check_postgres 可用于监测数据库健康并且报告不寻常的条件。check_postgres结合Nagios和MRTG， 但也可以独立运行。

PostgreSQL和其它数据库产品比较起来是低维护量的。 但是，适当在这些任务上放一些注意将更加能够确保我们的愉快工作和获取对这个系统富有成效的经验。

23.1. 日常清理

PostgreSQL数据库需要定期/维护被称为*vacuuming*。 很多安装足以通过*autovacuum*守护进程执行清理，正如Section 23.1.6所描述的。 你可能需要调整清理参数为你的情况得到最好的结果。一些数据库管理员会想 补充或取代手动管理 `VACUUM` 命令的进程活动， 这通常根据cron或者任务调度程序脚本执行的。 设置手动管理适当清理，理解下面几个子部分讨论的问题是必要的。依赖于autovacuuming的管理员可能仍然希望 浏览此材料来帮助他们理解和调整autovacuuming。

23.1.1. 清理基础

PostgreSQL的VACUUM命令 由于以下几个原因，必须周期性处理每个表：

1. 恢复那些由已更新或已删除的行占据的磁盘空间
2. 更新PostgreSQL查询规划器使用的数据统计信息。
3. 更新可见性映射，这加速了唯一索引扫描
4. 避免因为事务*ID*重叠造成的老数据丢失。

对上面每个原因进行 `VACUUM` 操作的频率和范围不同。正如下面 每个部分所述。

有 `VACUUM` 的两个变形：标准 `VACUUM` 和 `VACUUM FULL` 。 `VACUUM FULL` 可以回收更多 磁盘空间，但运行速度要慢得多。另外， `VACUUM` 的标准形式可以与生产 数据库操作并行运行。（命令 `SELECT` , `INSERT` , `UPDATE` 和 `DELETE` 将继续正常工作，当被清理的时候，但你使用诸如命令 `ALTER TABLE` 将不能够修改表的定义）。 `VACUUM FULL` 需要正运行的表上的排他锁，并且不能与其它表使用并行完成。一般地，因此， 管理员应该尽量使用标准的 `VACUUM` 避免 `VACUUM FULL` 。

另外， `VACUUM` 需要大量的I/O操作，可能导致其它活动中的会话性能严重降低。 调整配置参数以降低后端清理的性能影响— 参阅Section 18.4.4 获取更多信息。

23.1.2. 恢复磁盘空间

在正常的PostgreSQL操作里， 对一行的 `UPDATE` 或者 `DELETE` 并未立即删除旧版本的数据行。这个方法对于获取多版本并发控制的好处是必要的 (MVCC参阅Chapter 13)： 如果一个行的版本仍有可能被其它事务看到，那么你就不能删除它。 但到了最后，不会有任何事务对过期的或者已经删除的行感兴趣。 而它占据的空间必须为那些新行的使用而回收，以避免对磁盘空间需求无限的增长。 这件事是通过运行 `VACUUM` 实现的。

`VACUUM`的标准形式删除表中的死行以及索引，并且标记未来可重新使用的可用空间。 然而，它不会返回空间到操作系统， 除了在特殊情况下，其中表结尾的一个或多个页面完全自由，并且可轻易获得排它表锁。 相比之下， `VACUUM FULL`通过写入没有死表空间的表文件完整的新版本来压缩表， 这最大限度地减少了表的大小，但也需要相当长的时间。 这还需要表的新副本额外的磁盘空间，直到操作完成。

定期清理通常目标是执行标准`VACUUM`通常足以避免需要`VACUUM FULL`。 该自动清理后台程序试图以这种方式工作，而事实上 从未提出`VACUUM FULL`。在这种方法中，想法是不能保持表的最小尺寸但要保持磁盘空间用法稳定状态： 每个表占用的空间相当于其最小尺寸加上清理期间被用完的许多空间。 虽然`VACUUM FULL`可用于收缩表到其最小尺寸， 并返回该磁盘空间给操作系统，如果该表将来只是再次增长，那么毫无意义。 因此，比起为了维护更新频繁的表而很少运行`VACUUM FULL`来说， 运行适度频繁标准`VACUUM`是一个好的方法。

某些管理员倾向于定期清理自己，例如当负载较低的时候夜间做所有的工作。 按照固定的时间执行清理的困难是，如果一个表在更新活动中有意想不到的秒杀， 它可能膨胀，所以`VACUUM FULL`的确有必要回收空间。 使用自动清理后台程序解决了这个问题， 因为守护进程时间表清理动态响应更新活动。 完全禁用守护进程是不明智的，除非你 有一个可预测的工作量。一个可能的妥协是 设置守护进程的参数，这样只会反应异常沉重的更新活动，从而使事情变得不可收拾， 当负载是典型的，而预定的`VACUUM`希望做更多的工作。

对于那些不使用自动清理的，典型的做法是一旦在低使用率期间的一天安排数据库范围的`VACUUM`， 通过更新频繁的表更加频繁的清理作为必要补充。 （有些具有极高更新速率的安装每隔几分钟清理他们最繁忙的表）。 如果你在集群中有多个数据库，则不要忘了`VACUUM`；该程序vacuumdb可能会有所帮助。

> **Tip:** 普通`VACUUM`可能不尽如人意， 当一个表中包含大量的死行版本作为大规模更新或删除活动的结果。 如果你有这样一个表，并且你需要回收占用的多余磁盘空间， 则需要使用`VACUUM FULL`， 或者CLUSTER 或者ALTER TABLE的表重写变形之一。 这些命令重写表的全新副本，并建立新的索引。 所有这些选项都需要排它锁。需要注意的是他们也暂时使用额外的磁盘空间大致等于表的大小， 因为表的旧副本以及索引不能释放，直到新的完成。
>
> **Tip:** 如果你有一个表，它的内容经常被完全删除， 那么可以考虑用TRUNCATE而不是后面跟着`VACUUM`‘ 的`DELETE`。 `TRUNCATE`立即删除整个表的内容， 而不要求随后的`VACUUM`或者`VACUUM FULL`来恢复现在未使用的磁盘空间。 缺点是违反了严格的MVCC语义。

23.1.3. 更新规划器统计

PostgreSQL的查询规划器依赖一些有关表内容的统计信息用以为查询生成好的规划。 这些统计是通过ANALYZE命令获得的， 你可以直接调用这条命令，也可以把它当做`VACUUM`里的一个可选步骤来调用。拥有合理准确的统计是非常重要的，否则， 选择了恶劣的规划很可能降

低数据库的性能。

如果启用自动清理后台程序，将自动发出 `ANALYZE` 命令，当表的内容已经充分改变。 然而，管理员可能更愿意依靠手动安排的 `ANALYZE` 操作，尤其是 如果它是已知的表上的更新活动，不会影响 "感兴趣"列的统计。守护进程时间表 `ANALYZE` 严格作为插入或更新行数的函数; 它不知道是否这将导致有意义的统计变化。

和为了回收空间做清理一样，经常更新统计信息也是对更新频繁的表更有用。 不过，即使是更新非常频繁的表，如果它的数据的统计分布并不经常改变， 那么也不需要更新统计信息。一条简单的拇指定律就是想想表中字段的最大跟最小值改变的幅度。 比如，一个包含行更新时间的 `timestamp` 字段将随着行的追加和更新稳定增长最大值； 这样的字段可能需要比那些包含访问网站的URL的字段更频繁一些更新统计信息。 那些URL字段可能改变得一样频繁，但是其数值的统计分布的改变相对要缓慢得多。

我们可以在特定的表，甚至是表中特定的字段上运行 `ANALYZE`， 所以如果你的应用有需求的话，可以对某些信息更新得比其它信息更频繁。 不过，在实际中，通常最好只是分析整个数据库，因为它是一个快速操作。 `ANALYZE` 使用了统计学上的随机采样的方法进行行采样， 而不是把每一行都读取进来。

> **Tip:** 尽管用 `ANALYZE` 针对每个字段进行挖掘的方式可能不是很实用， 但你可能还是会发现值得针对每个字段对 `ANALYZE` 收集的统计信息的详细级别进行调整。 那些经常在 `WHERE` 子句里使用的字段如果有非常不规则的数据分布， 那么就可能需要比其它字段更细致的数据图表。 参阅 `ALTER TABLE SET STATISTICS` 。或者使用default_statistics_target 配置参数改变缺省数据库。
>
> 另外，默认情况下有选择性函数的有限信息可用。但是， 如果您使用函数调用创建一个表达式索引，有用的统计数据将 收集有关函数的信息，这样可以使用表达式索引大大提高查询规划。
>
> **Tip:** 该自动清理后台程序不会为外表发出 `ANALYZE` 命令， 因为它没有办法决定多长时间可能是有用的。如果您的查询需要外表的统计信息进行适当的规划， 在表上合适的时间运行手动管理 `ANALYZE` 命令是一个好主意。

23.1.4. 更新可见视图

清理保持可见视图为了每个表跟踪只包含元组的页面， 对所有活动事务可见（以及所有未来的事务，直至页面再次修改）。这有两个目的。首先，在下次运行时清理 本身可以跳过这些页面，因为没有什么可清理的。

其次，它允许PostgreSQL回答一些只使用索引，没有参考基础表的查询。 由于PostgreSQL索引不包含能见度信息元组， 普通索引扫描为每个匹配索引项抓取堆元组， 检查它是否由当前事务可见。另外一方面，索引扫描首先检查 能见度视图。如果它知道， 页面上的所有元组

是 可见的，可用忽略堆抓取。在大型数据集上这是最显著的。 其中可见视图可以防止磁盘访问。 可见视图远远比堆小，所以即使堆非常大，它可以很容易地缓存。

23.1.5. 避免事务ID重叠造成的问题

PostgreSQL的MVCC事务语意依赖于比较事务 ID(XID)的数值： 一条带有大于当前事务XID的插入XID的行版本是"属于未来的"， 并且不应为当前事务可见。但是因为事务ID的大小有限(在我们写这些的时候是32位)， 如果集群一次运行的时间很长(大于40亿次事务)，那么它就要受到事务*ID*重叠的折磨： XID计数器回到零位，然后突然间所有以前的事务就变成看上去是在将来的— 这意味着它们的输出将变得可见。 简而言之，可怕的数据丢失。实际上数据仍然在那里， 但是如果你无法获取数据，这么说也只是自我安慰罢了。 为了避免这种情况，有必要清理至少每二十亿事务的每个数据库中的每个表。

周期性的运行VACUUM可以解决这个问题的原因在于PostgreSQL 可以储存特殊的XID(`FrozenXID`)。这个XID不遵循普通XID比较规则， 总是被认为比任何普通的XID旧。 普通的XID使用模-2³¹算法进行比较。 这就意味着对于每个普通的XID， 总是有二十亿个XID是"更旧"以及二十亿个XID"更新"； 表达这个意思的另外一个方法是普通的XID 空间是没有终点的环。因此，一旦某行带着特定的普通XID创建出来， 那么该行将在以后的二十亿次事务中表现得是"在过去"，而不管我们说的是哪个普通XID。 如果该行在超过二十亿次事务之后仍然存在，那么它就会突然变成在将来的行。 为了避免数据丢失，老的行必须在到达二十亿次事务的年龄之前的某个时候赋予 `FrozenXID` 。 一旦它被赋予了这个特殊的XID ，那么它们在所有普通事务面前表现为"在过去"， 而不管事务ID是否重叠，因此这样的行不管保存多长时间，直到删除之前都会完好。 这个XID的重新赋值是 `VACUUM` 控制的。

vacuum_freeze_min_age 控制着在它之前更旧的XID将被替换为 `FrozenXID` 。 较大的设置值防止了事务信息变长， 较小的值增加了在表必须被清理之前可以清理事务的数量。

`VACUUM` 通常会忽略没有任何死行版本页面， 但这些页面可能仍然有旧XID值的行版本。为了确保所有旧的XID已被 `FrozenXID` 替换， 需要全表扫描。vacuum_freeze_table_age控制 `VACUUM` 的执行：为了 `vacuum_freeze_table_age` 减去 `vacuum_freeze_min_age` 事务，如果没有完全扫描整个表， 则将其设置为0，强制 `VACUUM` 总是扫描所有页面，有效地忽略可见视图。

表在清理之前允许执行的最大事务次数 是20亿事务减去 `VACUUM` 上次扫描整个表时的 `vacuum_freeze_min_age` 值。 如果超过这个限制就很可能造成数据丢失。为了保证数据安全， 必须在任何可能包含旧于autovacuum_freeze_max_age指定的XID的 表上调用autovacuum。甚至在autovacuum被禁用的情况下也可以调用。

这就意味着，一个未被清理的表将会在大约 `autovacuum_freeze_max_age` 减去 `vacuum_freeze_min_age` 次事务后被自动清理。 对于那些周期性清理以回收空间的表来说，这个并不重要。 对于静态表(包括只插入不更新/删除的表)，因为不需要回收空间的清理， 所以可以尝试最大化强制清理的时间间隔， 也就是增加 `autovacuum_freeze_max_age` 的值或 减少 `vacuum_freeze_min_age` 的值。

`vacuum_freeze_table_age` 有效最大值是0.95* `autovacuum_freeze_max_age`；高于它的设置将覆盖最大值。高于 `autovacuum_freeze_max_age` 的值是没有意义的， 因为自动清理将在这一点被触发，在这发生之前，0.95乘数留下一些空间来执行手动 `VACUUM` 。作为一个经验法则， `vacuum_freeze_table_age` 应设置为稍微低于 `autovacuum_freeze_max_age` 的一个值。 留出足够的空隙，以便定期安排 `VACUUM` 或通过运行在该窗口中的正常删除和更新活动触发自动清理。 将其设置得接近可能导致抗回绕自动清理， 即使表最近被清理以回收空间，而较低的值会导致更多频繁的全表扫描。

增加 `autovacuum_freeze_max_age` 以及 `vacuum_freeze_table_age` 的唯一不利之处在于数据库集群的 `pg_clog` 子目录将会占用更多空间， 因为它必须为所有 `autovacuum_freeze_max_age` 之后的事务存储提交状态。 每个事务提交状态使用2字节，因此如果 `autovacuum_freeze_max_age` 设置为最大允许值为20亿， `pg_clog` 将会增加到大约500M。 如果这个尺寸比起你的数据库来只是小菜一碟，我们推荐你将 `autovacuum_freeze_max_age` 设为允许的最大值。否则， 如何设置将取决于你愿意给 `pg_clog` 多大的空间。默认值是2亿， 大约需要50MB的 `pg_clog` 存储空间。

减小 `vacuum_freeze_min_age` 的不利之处是可能导致 `VACUUM` 做无用功： 如果行在不久之后就被修改，那么将XID修改为 `FrozenXID` 就是在浪费时间， 因为它很快就将获得一个新的XID。因此这个设置应当足够大以使得行不被过早的冻结。 减小 `vacuum_freeze_min_age` 的另一个不利之处 是事务插入或修改行的准确细节将会很快丢失。 这个信息有时迟早会派上用场， 特别是数据库失败之后分析究竟发生了什么错误的时候。 因为这两个原因，在完全静态的表上减小这个值是不明智的。

为了跟踪数据库中最老的XID寿命， `VACUUM` 在系统表 `pg_class` 和 `pg_database` 里存储了XID统计。 尤其是一个数据库的 `pg_class` 行中的 `relfrozenxid` 字段 包含了最后一个整表 `VACUUM` 命令使用的冻结终止XID。 系统保证在该表中所有比这个终止XID老的普通XID都被 `FrozenXID` 代替。 同样，一个数据库的 `pg_database` 行中的 `datfrozenxid` 字段是普通XID的下界— 它只是数据库中每个表 `relfrozenxid` 的最小值。 检查这个信息的一个便利方法是执行下面的查询：

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

`age` 字段用于测量从中止XID到当前事务XID的数目。

`VACUUM` 常常只扫描自上次清理已被修改的页， 但 `relfrozenxid` 仅仅提前扫描整个表。 当 `relfrozenxid` 大于 `vacuum_freeze_table_age` 事务时，当使用 `VACUUM` 的 `FREEZE` 选项时，或者当所有页需要清理删除死行版本，进行全表扫描。 当 `VACUUM` 扫描全表时，

`age(relfrozenxid)` 应当立即使用稍微大于 `vacuum_freeze_min_age` 的值 (比 `VACUUM` 启动之后开始的事务数目稍大)。如果在表上提出非全表扫描 `VACUUM` 直到 超过 `autovacuum_freeze_max_age` ， 则将会很快在表上强制进行自动清理。

如果从表中清理旧XID失败，那么当数据库的旧XID到达1000万以后， 系统将发出类似下面这样的警告信息：

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

手动 `VACUUM` 应该修复这个问题，正如提示建议；但是注意 `VACUUM` 必须通过超级用户执行，否则无法处理系统目录，并且不能提高数据库的 `datfrozenxid` 。 如果忽略了上面的警告信息，那么系统将在距离重叠小于100万次的时候关闭， 并且拒绝开始任何新的事务：

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "myd
HINT:  Stop the postmaster and use a standalone backend to VACUUM in "mydb".
```

这个100万的事务安全边界留下来用于让管理员在不丢失数据的情况下进行恢复， 方法是手工执行所需要的 `VACUUM` 命令。不过，因为一旦进入了安全关闭模式， 系统就不能再执行命令，做这件事情的唯一的方法是停止主服务器， 使用一个单独运行的后端来执行 `VACUUM` 。关闭模式不会强制于独立运行的后端。 参阅postgres手册获取有关使用独立运行后端的细节。

23.1.6. Autovacuum守护进程

PostgreSQL带有一个可选高度推荐的特性 叫做*autovacuum*守护进程， 它的目的是自动执行 `VACUUM` 和 `ANALYZE` 命令。 在打开这个选项之后，autovacuum守护进程将检查那些有大量插入、 更新、删除行操作的表。这些检查使用行级别的统计收集设施；因此，除非把track_counts设置为 `true` ， 否则无法使用autovacuum守护进程。 在缺省配置下， 启用autovacuum守护进程并且合理设置相关配置参数。

该"自动清理后台程序"实际上是由多个进程组成的。 有一个持久守护进程，称为*autovacuum launcher* 它是负责为所有数据库启动*autovacuum worker*进行。 该发射器将分发工作跨越时间， 每个数据库内每autovacuum_naptime秒内尝试启动1个工作。 （因此，如果安装有 `_N_` 个数据库，每 `autovacuum_naptime` / `_N_` 秒将开始一个新的。） 最多autovacuum_max_workers工作进程在同一时间允许运行。 如果正在处理多于 `autovacuum_max_workers` 的数据库，一旦第一个处理完成将处理 下一个数据库。每个工作进程将检查它的数据库中的每个表，并且 执行 `VACUUM` 和/或者按需要执行 `ANALYZE` 。 使用 `log_autovacuum_min_duration` 可以监控 自动清理活动。

如果在很短的时间中需要清理若干个大表， 则 所有自动清理的工人可能需要很长一段时间清理这些表。 这将导致其它表和数据库不能被清理，直到工人可用。 在单一的数据库中有多少人可能没有限制， 但尽量避免已经被其他人完成的重复工作。需要注意的是运行数 不计

入max_connections或者 superuser_reserved_connections限制。

那些 `relfrozenxid` 大于autovacuum_freeze_max_age 的表将总是被清理（这也适用于通过存储参数修改的冻结最大时间的那些表；参见下文）。 否则，如果上次 `VACUUM` 之后的过期行的数量超过了"清理阈值"， 那么就清理该表。清理阈值定义为:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

这里的清理基本阈值是autovacuum_vacuum_threshold， 清理缩放系数是autovacuum_vacuum_scale_factor， 行数是 `pg_class` . `reltuples` ， 过期行的数量是从统计收集器里面获取的， 这是一个半精确的计数，由每次 `UPDATE` 和 `DELETE` 操作更新。 半精确的原因是在重负载时有些信息可能会丢失。 如果表的 `relfrozenxid` 值大于 `vacuum_freeze_table_age` ， 扫描整个表 冻结旧元组，并且提升 `relfrozenxid` ，否则仅仅扫描上次清理后修改的页。

为了分析，使用了一个类似的条件：分析阈值，定义为：

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

它会和上次 `ANALYZE` 插入、更新、删除的总行数进行比较。

临时表不能被自动清理进行访问。因此，适当的清理和分析操作应通过会话SQL命令执行。

缺省的阈值和伸缩系数是从 `postgresql.conf` 里面取得的，不过， 它可能基于表而覆盖。参阅存储参数获取更多细节。 如果通过存储参数已经改变设置，那么则使用该值；否则使用全局设置。 参阅Section 18.10获取有关全局设置的更多细节。

除了基本阈值和缩放系数之外， 还有6个autovacuum 参数可以通过存储参数为每个表进行设置。 第一个参数， `autovacuum_enabled` 可以设置为 `false` 让autovacuum 守护进程完全忽略某个表。这种情况下，autovacuum只有在为了避免事务ID 重叠必须清理整个数据库的时候才会动那个表。接下来两个参数， `autovacuum_vacuum_cost_delay` 和 `autovacuum_vacuum_cost_limit` 用于针对特定的表为基于开销的清理延迟特性设置数值。参阅Section 18.4.4。 `autovacuum_freeze_min_age` ， `autovacuum_freeze_max_age` 和 `autovacuum_freeze_table_age` 分别为vacuum_freeze_min_age, autovacuum_freeze_max_age和 vacuum_freeze_table_age设置数值。

当多个工作者正在运行，成本限制在所有正在运行的人中是"balanced"， 从而使系统上的总影响是相同的，而不管实际运行人数。

23.2. 经常重建索引

有时候我们值得用REINDEX命令周期性重建索引。

已经完全空的B树索引页会回收重新使用。然而，这可能是空间使用低效：如果所有，但页面上的几个索引键已经被删除而页面仍分配。因此，使用模式其中大多数，但不是全部，最终被删除的每个范围内的键将看到空间低效使用。对于这样的使用模式，推荐周期性重建索引。

对于非B-tree索引的膨胀潜能可能还没有很好地分析。 在使用非B-tree索引的时候保持对索引的物理尺寸的周期性监控是个很好的主意。

还有，对于B-tree索引，一个新建立的索引从某种意义上比更新了多次的访问起来稍微要快，因为在新建立的索引上，逻辑上连接的页面通常物理上也连接在一起 (这样的考虑目前并不适用于非B-tree索引)。仅仅从提高访问速度角度出发， 可能我们也值得周期性的重建索引。

23.3. 日志文件维护

把数据库服务器的日志输出保存在一个地方而不是仅仅把它们放到 `/dev/null` 里是个好主意。 在碰到危险的时候，日志输出是非常宝贵的。不过， 日志输出可能很庞大(特别是在比较高的调试级别上)， 而且你不会无休止地保存它们。你需要滚动日志文件， 这样生成新的日志文件并且经常抛弃老的。

如果你简单地把 `postgres` 的stderr重定向到一个文件中， 你会有日志输出，但是截断日志文件的唯一的方法是停止并重启主服务器。 这样做对于PostgreSQL开发环境中是可以的， 但是你肯定不想在生产环境中也这么干。

一个更好的办法是把主服务器的stderr输出发送到某种日志滚动程序里。 我们有一个内置的日志滚动程序，你可以通过在 `postgresql.conf` 里设置配置参数 `logging_collector` 为 `true` 的办法打开它。 这个程序的控制参数在Section 18.8.1里描述。 你也可以使用这个方法捕获机器上可读CSV(逗号分隔值)格式的日志数据。

另外，如果你准备使用其他服务器软件，你可能更喜欢 使用一个外部日志滚动程序(比如PostgreSQL中 Apache附带的rotatelogs工具)， 为了做到这一点， 你可以将stderr的输出重定向到所需程序。如果你用 `pg_ctl` 启动服务器， 那么stderr已经重定向到stdout，因此你只需要一个管道命令，比如：

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

另外一种生产级的管理日志输出的方法就是把它们发送给syslog并且让syslog 处理滚动。要利用这个工具，我们需要设置 `postgresql.conf` 里的 `log_destination` 为 `syslog` (记录syslog日志)。 然后在你想强迫syslog守护进程开始写入一个新日志文件的时候， 就可以发送一个 `SIGHUP` 信号给它。如果你想自动滚动日志文件，那么我们可以配置 logrotate程序处理syslog的日志文件。

不过，在很多系统上，syslog不是非常可靠，特别是在大型日志信息的情况下； 它可能在你最需要那些信息的时候截断或者丢弃它们。还有，在Linux上， syslog会把每个消息刷新到磁盘上，导致很低下的性能。 你可以在syslog配置文件里面的文件名开头使用"`-`"来关闭这个行为。

请注意上面描述的所有解决方案关注的是在可配置的间隔上开始一个新的日志文件， 它们并没有删除不再需要的旧日志文件。你可能还需要设置一个批处理， 周期地删除旧日志文件。另外一个可能的解法是配置日志滚动程序， 让它周期地覆盖旧的日志文件。

pgBadger 是一个外部项目，做复杂的日志文件分析。 当重要信息出现在日志文件中时，check_postgres 提供了Nagios警告，以及许多其他特殊条件的检测。

Chapter 24. 备份与恢复

Table of Contents

和任何包含珍贵数据的东西一样，PostgreSQL数据库也应该经常备份。 尽管这个过程相当简单，但是我们还是应该理解做这件事所用的一些技巧和假设。

- SQL转储
- 文件系统级别备份
- 在线备份

每种备份都有自己的优点和缺点。在下面的章节中依次进行讨论。

24.1. SQL转储

SQL转储的方法是创建一个文本文件，里面都是SQL命令， 当把这个文件回馈给服务器时，将重建与转储时状态一样的数据库。 PostgreSQL为这个用途提供了pg_dump工具。 这条命令的基本用法是：

```
pg_dump _dbname_ > _outfile_
```

正如你所见，pg_dump把结果输出到标准输出。 我们下面就可以看到这样做有什么好处。

pg_dump是一个普通的PostgreSQL 客户端应用(尽管是个相当聪明的东西)。 这就意味着你可以从任何可以访问该数据库的远端主机上面进行备份工作。 但是请记住pg_dump不会以任何特殊权限运行。具体说来， 就是它必须要有你想备份的表的读权限，因此， 实际上你几乎总是要成为数据库超级用户。

要声明pg*dump*应该以哪个用户身份进行连接， 使用命令行选项`-*h``_host* 和 -p `` `_port_ `` 。缺省主机是本地主机或环境变量 `PGPORT` 声明的值。类似的，缺省端口是环境变量 `PGPORT` 或(如果它不存在的话)编译好了的缺省值。服务器通常都有相同的缺省， 所以还算方便。

和任何其它PostgreSQL客户端应用一样，pg_dump缺省时用 与当前操作系统用户名同名的数据库用户名进行连接。 要覆盖这个名字，要么声明 `-U` 选项， 要么设置环境变量 `PGUSER` 。 请注意pg_dump 的连接也和普通客户应用一样要通过客户认证机制(在Chapter 19里描述)。

pg_dump超过后边描述的其它备份方法的一个重要优点 是pg_dump的输出通常可以 重新载入PostgreSQL新版本， 然而文件级别备份和连续归档都因 服务器版本而异。pg_dump是 将传输数据库到另一台机器体系结构工作时唯一的方法， 如从32位变到64位服务器。

由pg_dump创建的备份在内部是一致的， 也就是说，在pg_dump运行的时候转储的是数据库的快照。 pg_dump工作的时候并不阻塞其它的对数据库的操作 (但是会阻塞那些需要排它锁的操作，比如 `ALTER TABLE`)。

> **Important:** 如果你的数据库结构依赖于OID(比如说用做外键)， 那么你必须告诉pg_dump把OID也导出来。要导出OID， 可以使用 `-o` 命令行选项。

24.1.1. 从转储中恢复

```
psql _dbname_ < _infile_
```

这里的 `_infile_` 就是通过pg*dump*命令的文件输出。 这条命令不会创建`_*dbname* `数据库， 你必须在执行psql前自己从` template0 `创建(也就是用` createdb -T template0 `` _dbname_ `` 命令)。 psql支持类似pg_dump的选项用以控制数据库服务器位置和用

户名。 参阅psql的手册获取更多信息。

在开始运行恢复之前，目标库和所有在转储出来的库中拥有对象的用户， 以及曾经在某些对象上被赋予权限的用户都必须已经存在。如果这些不存在， 那么恢复将失败，因为恢复过程无法把这些对象恢复成原有的所有权和/或权限。 有时候你希望恢复权限，不过通常你不需要这么做。

缺省时，psql脚本将在遇到错误的时候仍然继续执行。 你可能希望运行带有 `ON_ERROR_STOP` 变量设置的 psql以改变操作，并且如果发生SQL错误则带有 退出状态码3的psql退出。

```
psql --set ON_ERROR_STOP=on dbname < infile
```

不管上述哪种方法都只能得到部分恢复了的数据库。另外， 你可以将整个恢复过程当成一个单独的事务，这样就能够保证要么全部恢复成功， 要么全部回滚。可以通过向psql传递 `-1` 或者 `--single-transaction` 命令行 参数达到此目的。使用这个模式的时候即使一个很微小的错误也将导致已经运行 了好几个小时的恢复过程回滚。尽管如此，这种模式也比手动清除哪些不完整的 恢复数据强。

pg_dump和psql可以通过管道读写， 这样我们就可能从一台主机上将数据库目录转储到另一台主机上，比如：

```
pg_dump -h _host1_ _dbname_ | psql -h _host2_ _dbname_
```

> **Important:** pg_dump生成的转储输出是相对于 `template0` 的。 这就意味着任何加入到 `template1` 的语言、 过程等都会经由pg_dump转储。 这样在恢复的时候，如果你使用的是自定义的 `template1`， 那么你必须从 `template0` 中创建空的数据库， 就像我们上面的例子那样。

一旦完成恢复，在每个数据库上运行ANALYZE是明智的举动， 这样优化器就有可用的统计数据了。 Section 23.1.3 和Section 23.1.6获取更多信息。 关于如何有效加载海量数据到PostgreSQL的更多信息， 参考Section 14.4。

24.1.2. 使用pg_dumpall

pg_dump在一个时间只转储一个单独的数据库， 它不转储有关角色或表空间信息（因为这些是集群范围，而不是每个数据库）。 为了支持 方便转储整个数据库集群的全部内容。 因此我们提供了pg_dumpall程序。 pg_dumpall备份一个给出的集群中 的每个数据库，同时还确保保留像角色和表空间这样的全局数据状态。 这个命令的基本用法是：

```
pg_dumpall > _outfile_
```

生成的转储可以用psql恢复：

```
psql -f _infile_ postgres
```

实际上，你可以声明任意现有的数据库进行连接， 但是如果你是向一个空的数据库集群装载， 那么`postgres`应该是比较好的选择。 恢复pg_dumpall的转储的时候通常需要数据库超级用户权限， 因为我们需要它来恢复角色和表空间信息。如果使用了表空间， 需要注意转储中的表空间路径必须适合新的安装。

pg_dumpall的工作原理是发射命令来重新创建角色， 表空间和空数据库，然后为每个数据库调用pg_dump。 这意味着，虽然每个数据库内部一致， 但不同的数据库快照可能不是恰好同步。

24.1.3. 处理大数据库

当创建大的pg_dump输出文件时， 限制产生问题的一些操作系统允许最大文件大小。 因为pg_dump输出到标准输出， 你可以用标准的Unix工具绕开这个问题：有一些可能的方法：

使用压缩转储. 使用你熟悉的压缩程序(比如gzip)：

```
pg_dump _dbname_ | gzip > _filename_.gz
```

使用下面命令恢复：

```
gunzip -c _filename_.gz | psql _dbname_
```

或者：

```
cat _filename_.gz | gunzip | psql _dbname_
```

使用 **`split`** . `split`允许用下面的方法把输出分解成操作系统可以接受的大小。 比如，让每个块大小为1MB：

```
pg_dump _dbname_ | split -b 1m - _filename_
```

用下面命令恢复：

```
cat _filename_* | psql _dbname_
```

使用**pg_dump**自定义转储格式. 如果PostgreSQL是在一个安装了zlib 压缩库的系统上制作的， 那么自定义转储格式将在写入输出文件的时候压缩数据。 它会生成和使用`gzip`类似大小的转储文件，但是还附加了一个优点： 你可以有选择地恢复库中的表。下面的命令用自定义转储格式转储一个数据库：

```
pg_dump -Fc _dbname_ > _filename_
```

自定义格式的转储不是脚本，不能用于psql，而是需要使用pg_restore转储。 比如：

```
pg_restore -d _dbname_ _filename_
```

请参阅pg_dump和pg_restore手册获取细节。

对于非常大的数据库，你可能需要结合 `split` 以及其他两种方法之一。

使用**pg_dump**的并行转储功能. 为了加快大数据库的转储，你可以使用pg_dump并行模式。这将同时转储多个表。你可以使用 `-j` 参数控制并行性程度。 并行转储只支持"目录"归档模式。

```
pg_dump -j _num_ -F d -f _out.dir_ _dbname_
```

你可以使用 `pg_restore -j` 并行恢复转储。这将为任何"自定义"或者 "目录"归档模式工作，是否它已经使用 `pg_dump -j` 创建。

24.2. 文件系统级别备份

另一个备份的策略是直接拷贝PostgreSQL用于存放数据库数据的文件。 我们在Section 17.2里解释了这些文件的位置， 你可以用自己喜欢的任何常用文件系统备份的方法， 例如：

```
tar -cf backup.tar /usr/local/pgsql/data
```

不过，你要受到两个限制，令这个方法不那么实用，或者至少比pg_dump的方法逊色一些：

1. 为了进行有效的备份，数据库服务器必须被关闭。 像拒绝所有连接这样的折衷的方法是不行的， （部分因为 `tar` 和类似的工具在做备份的时候并不对文件系统的状态做原子快照。 但也因为服务器内部缓冲数据）。 有关关闭服务器的信息可以在Section 17.5里面找到。 不用说，你在恢复数据之前， 同样必须关闭服务器。

2. 如果你曾经深入了解了数据库在文件系统布局的细节， 你可能试图从对应的文件或目录里备份几个表或者数据库。 这样做是没用的，因为包含在这些文件里的信息只是部分信息。 还有一半信息在提交日志文件 `pg_clog/*` 里面， 它包含所有事务的提交状态。 只有拥有这些信息，表文件的信息才是可用的。当然， 试图只恢复表和相关的 `pg_clog` 数据也是徒劳的， 因为这样会把数据库集群里的所有其它没有用的表的信息都拿出来。 所以文件系统的备份只适用于一个数据库集群的完整恢复。

另外一个文件系统备份的方法是给数据目录做一个"一致的快照"， 条件是文件系统支持这个功能(并且你愿意相信它是实现正确的)。 典型的过程是制作一个包含数据库的卷的"冻结快照"， 然后把整个数据库目录(不仅仅是部分， 见上文)从快照拷贝到备份设备， 然后释放冻结快照。这样甚至在数据库服务器在运行的时候都可以运转。 不过，这样创建的备份会把数据库文件保存在一个没有恰当关闭数据库服务器的状态下； 因此，如果你在这个备份目录下启动数据库服务器， 它就会认为数据库服务器经历过崩溃并且重放WAL日志。 这不是个问题，只要意识到它即可(并且确信在自己的备份中包含WAL文件)。 在采用快照减少恢复时间之前，你可以执行 `CHECKPOINT` 。

如果你的数据库分布在多个文件系统上， 那么可能就没有任何方法获取所有卷上准确的同步冻结快照。比如， 你的数据文件和WAL日志在不同的磁盘上，或者表空间在不同的文件系统上， 这种情况下就不可能使用快照，因为快照必须是同时的。 在你信任这样的情况下的一致性快照的技术之前，仔细阅读你的文件系统文档。

如果同步快照是不可能的，该选项关闭数据库服务器足够长的时间来建立所有冰冻的快照。 另一种选择是执行一个连续归档基础备份(Section 24.3.2) 因为这样的备份在备份过程中免于文件系统变化。这需要在备份过程中启动连续归档。 恢复是通过使用连续存档恢复(Section 24.3.4)完成的。

另外一个选择是使用rsync执行一次文件系统备份。 这是通过在数据库服务器正在运行的时候运行第一次rsync， 然后关闭数据库服务器一段足够的时间长度，用于运行第二次rsync。 第二次rsync会比第一次快很多，因为它要传输的数据相对较少， 并且最后的结果是一致的，因为服务器已经停止运行了。这个方法允许用很少的时间执行一次文件系统备份。

需要注意的是文件系统备份往往比SQL转储大。 比如pg_dump不用导出索引， 只是创建它们的命令。然而，文件系统备份可能会更快。

24.3. 在线备份以及即时恢复(PITR)

在任何时候，PostgreSQL都在集群的数据目录的`pg_xlog/`子目录里维护着一套预写日志(WAL)。 这些日志记录着每一次对数据库的修改细节。 这些日志存在是为了防止崩溃： 如果系统崩溃，数据库可以通过"重放"上次检查点以来的日志记录以恢复数据库的完整性。 但是，日志的存在让它还可以用于第三种备份数据库的策略： 我们可以组合文件系统备份与WAL文件的备份。如果需要恢复，我们就恢复备份， 然后重放备份了的WAL文件，把备份恢复到当前的时间。这个方法对管理员来说， 明显比以前的方法更复杂，但是有非常明显的优势：

- 在开始的时候我们不需要一个非常完美的一致的备份。 任何备份内部的不一致都会被日志重放动作修改正确 (这个和崩溃恢复时发生的事情没什么区别)。因此我们不需要文件系统快照的功能， 只需要tar或者类似的归档工具。
- 因为我们可以把无限长的WAL文件序列连接起来， 所以连续的备份简化为连续地对WAL文件归档来实现。 这个功能对大数据库特别有用，因为大数据库的全备份可能并不方便。
- 我们可没说重放WAL记录的时候我们必须重放到结尾。 我们可以在任意点停止重放，这样就有一个在任意时间的数据库一致的快照。 因此，这个技术支持即时恢复： 我们可以把数据库恢复到你开始备份以来的任意时刻的状态。
- 如果我们持续把WAL文件序列填充给其它装载了同样的基础备份文件的机器， 我们就有了一套热备份系统：在任何点我们都可以启动第二台机器， 而它拥有近乎当前的数据库拷贝。

> **Note:** pg*dump*和 *pg_dumpall*没有产生文件系统级别备份，并且不能作为 连续归档解决方案的一部分。比如备份是符合逻辑的_并且不包含 WAL重放使用的足够信息。

和简单的文件系统备份技术一样，这个方法只能支持整个数据库集群的恢复， 而不是一个子集。同样，它还要求大量的归档存储：基础备份量可能很大， 而且忙碌的系统将生成许多兆需要备份的的WAL流量。但是， 它仍然时在需要高可靠性的场合下的最好的备份技术。

要想从连续归档中成功恢复（也被许多数据库供应商称为"在线备份"）， 你需要一套连续的WAL归档文件， 它们最远回朔到你开始备份的时刻。因此，要想开始备份， 你应该在开始第一次基础备份之前根据我们讨论过的归档WAL文件机制设置并测试你的步骤。

24.3.1. 设置WAL归档

抽象来看，一个运行着的PostgreSQL系统生成一个无限长的WAL日志序列。 系统物理上把这个序列分隔成WAL段文件，通常每段16M(在编译PostgreSQL的 时候可以改变其大小)。这些段文件的名字是数值命名的，这些数值反映他们在抽取出来的 WAL 序列中的位置。在不适用WAL归档的时候，系统通常只是创建几个段文件然 后"循环"使用它们，方法是把不再使用的段文件的名字重命名为更高的段编号。 系统假设那些内容比前一次检查点更老的段文件已经没用了，然后就可以循环利用。

在归档WAL数据的时候，我们希望在每个段文件填充满之后捕获之， 并且把这些数据在段文件被循环利用之前保存在某处。 根据应用以及可用的硬件的不同，我们可以有许多不同的方法"把数据保存在某处"： 我们可以把段文件拷贝到一个 NFS 挂载的目录，把它们放到另外一台机器上， 或者把它们写入磁带机里(需要保证你有办法把文件恢复为原名)，或者把它们打成包， 烧录到 CD 里，或者是其它的什么方法。为了给数据库管理员提供最大可能性的灵活性， PostgreSQL试图不对如何归档做任何假设。取而代之的是， PostgreSQL让管理员声明一个shell命令执行来拷贝一个完整的段文件到它需要去的地方。 该命令可以简单得就是一个 `cp` ，或者它可以调用一个复杂的shell脚本— 这些都由管理员决定。

为了启动WAL归档，设置wal_level 配置参数到 `archive` (或者 `hot_standby`), archive_mode 为 `on` ， 并且所使用的shell命令由配置参数archive_command声明， 它实际上总是放在 `postgresql.conf` 文件里的。在 `archive_command` 中， 任何 `%p` 都被要归档文件的绝对路径代替，而任何 `%f` 只是被文件名代替。 如果你需要在命令里嵌入一个真正的 `%` 字符，那么必须双写(`%%`)。 最简单的有用命令类似下面这样：

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

它将把WAL段拷贝到 `/mnt/server/archivedir` 目录。 这个只是一个例子，并非我们建议的方法， `%p` 和 `%f` 参数被取代之后，实际执行的命令看起来这样：

```
test ! -f /mnt/server/archivedir/00000001000000A900000065 && cp pg_xlog/00000001000000A90
```

为每一个归档的新文件产生类似的命令。

归档命令将在运行PostgreSQL服务器的同一个用户的权限下执行。 因此被归档的WAL文件实际上包含你的数据库里的所有东西， 所以你应该确保自己的归档数据不会被别人窥探；比如， 归档到一个没有组或者全局读权限的目录里。

有一点很重要：当且仅当归档命令成功时，它才返回零。 在得到一个零值结果之后， PostgreSQL将假设该WAL段文件已经成功归档， 因此它稍后将被删除或者被新的数据覆盖。 但是， 一个非零值告诉PostgreSQL该文件没有被归档； 因此它会周期性的重试直到成功。

归档命令通常应该设计成拒绝覆盖已经存在的归档文件。 这是一个非常重要的安全特性，可以在管理员操作失误 (比如把两个不同的服务器的输出发送到同一个归档目录)的时候保持你的归档的完整性

我们建议你首先要测试你准备使用的归档命令，以保证它实际上不会覆盖现有的文件， 并且在这种情况下它返回非零状态。 上边Unix例子中的命令包含单独的 `测试` 步骤。在一些Unix平台上， `cp` 有可以使用的开关比如 `-i` 可以做同样的简单的事情。 但是你不应该依靠这些而不验证返回的正确退出状态。 （尤其是，当使用 `-i` 并且已经存在目标文件时，GNU `cp` 将返回状态零， 这不是期望的操作。）

在设计你的归档环境的时候，请考虑一下如果归档命令不停失败会发生什么情况， 因为有些方面要求操作者的干涉，或者是归档空间不够了。比如， 如果你往磁带机上写，但是没有自动换带机，那么就有可能发生这种情况； 如果磁带满了，那就除非换磁带，否则啥事也做不了。 你应该确保任何错误条件或者要求操作员干涉的错误都会正确报告， 这样才能迅速解决这些问题。否则 `pg_xlog/` 目录会不停地填充WAL段文件， 直到问题解决。（如果文件系统由 `pg_xlog/` 填充，PostgreSQL将执行 PANIC关机。没有提交的事务将丢失，但是数据库将保持未连接直到你释放一些空间）。

归档命令的速度并不要紧，只要它能跟上你的服务器生成WAL数据的平均速度即可。 即使归档进程落在了后面一点，正常的操作也会继续进行。如果归档进程慢很多， 就会增加灾难发生时的数据丢失量。同时也意味着 `pg_xlog/` 目录包含大量未归档的日志段文件，并且可能最后超出了磁盘空间。 我们建议你监控归档进程，确保它是按照你的意识运转的。

在写自己的归档命令的时候，你应该假设被归档的文件最多 64个字符长并且可以包含ASCII字母、数字、点的任意组合。 我们不必要记住原始的全路径(`%p`)，但是有必要记住文件名(`%f` 。

请注意尽管WAL归档允许你恢复任何对PostgreSQL数据库的修改， 在最初的基础备份之后，它还是不会恢复对配置文件的修改 (`postgresql.conf` , `pg_hba.conf` 和 `pg_ident.conf`)， 因为这些文件都是手工编辑的，而不是通过SQL操作来编辑的。 所以你可能会需要把你的配置文件放在一个日常文件系统备份过程即可处理到的地方。 参阅Section 18.2获取如何重定位配置文件的知识。

因为归档命令仅在已经完成的WAL段上调用，因此， 如果你的服务器只产生很小的WAL流量或段之间的间隔很长， 那么在事务完成之后与其被安全归档之间就会存在很长的延时。 为了限制未归档数据的最长期限，可以设置archive_timeout强制服务器在切换 WAL段之间的时间间隔。需要注意的是， 由于强制切换而提早结束的已归档文件的大小与完整的归档文件相同。 因此将 `archive_timeout` 设为一个很小的值是不明智的— 它将很快耗尽归档空间。 将 `archive_timeout` 设置为60秒左右通常是比较合理的。

同样，如果你想确保刚刚完成的事务被立即归档， 那么也可以通过 `pg_switch_xlog` 手动强制切换段文件。 其它与WAL管理相关的工具函数在Table 9-60中列出。

当 `wal_level` 是 `minimal` 的时候，一些SQL命令进行优化以避免WAL日志， 正如Section 14.4.7所描述的。 如果在这些语句之一执行过程中打开归档或者流复制，WAL不包含归档恢复的足够信息。 (崩溃恢复不受影响)，出于这些原因，在服务器开始改变 `wal_level` 。 然而，重新加载配置文件时改变 `archive_command` 。 如果你希望暂时停止归档，这样做的一个方法是设置 `archive_command` 为空字符串 (`''`)。这将导致WAL文件在 `pg_xlog/` 中积累直到 `archive_command` 重新建立。

24.3.2. 进行一次基础备份

执行基础备份最简单方式是使用pg_basebackup工具， 它使用普通文件或者tar归档创建基础备份。 如果比pg_basebackup提供更多的灵活性，你也可以使用 低水平API（参阅Section 24.3.3）创建基础备份。

不必担心基础备份需要大量时间。然而， 如果你正常运行禁用了 `full_page_writes` 的服务器， 当运行备份时，你可能注意到性能方面， 因为 `full_page_writes` 有效强加在备份方式中。

要使用这个备份，你需要保存所有备份开始以及之后的WAL段文件。 为了帮助你实现这个任务，基础备份过程创建一个备份历史文件， 它马上存储到WAL归档区域。这个文件的名字是以你在使用文件系统备份的时候需要的第一个 WAL段文件的名字命名的。比如，如果开始WAL 文件是 `0000000100001234000055CD` ， 那么备份历史文件将命名为类似 `0000000100001234000055CD.007C9330.backup` 这样的东西。 这个文件名的第二部分表示在该WAL文件里面的准确位置，通常可以被忽略。 一旦你安全地把这些日志段文件归了档， 那么你就可以删除所有那些数值名字在这个文件前面的归档的WAL段。 文件系统备份不再需要它们了。当然，你应当保留几套备份以绝对确保可以恢复先前的数据。

备份历史文件只是一个小的文本文件。它包含你给予pg_basebackup的标签字符串， 以及备份的起始时间和终止时间。如果你使用这个标签来表示转储文件放在哪里， 则在需要的时候，归档的历史文件就足够告诉你转储文件存放在哪里了。

因为你必须保留直到最后一次基础备份的所有归档的WAL文件， 那么两次基础备份之间的间隔通常是根据你想在归档WAL 文件上花多少存储空间来定的。 你还应该考虑你准备在恢复上花多少时间。如果需要恢复的话— 系统将需要重放所有那些段， 而如果最后一次基础备份以来，时间已经很长了， 那么那些动作可能会花掉好些时间。

24.3.3. 使用低级别API进行基础备份

使用包含多个步骤的低水平API而不是pg_basebackup方法， 但是相对简单。在序列中执行这些步骤非常重要，并且在进行下一步之前需要验证当前步成功。

1. 确保WAL归档打开并且可以运转。

2. 以数据库超级用户身份连接到数据库，发出命令:

   ```
   SELECT pg_start_backup('label');
   ```

   这里的 `label` 是任意你想使用的这次备份操作的唯一标识 (一个好习惯是使用备份转储文件的放置地全路径)。 `pg_start_backup` 用备份信息在集群目录里 创建一个备份标签文件 `backup_label` 。 包含起始时间和标签字符串。该文件对于备份完整性是非常重要的，你需要从中恢复。

   至于你连接到集群中的那个数据库没什么关系。 你可以忽略函数返回的结果；但是如果它报告错误， 那么在继续之前先处理它。

   默认情况下， `pg_start_backup` 可能需要很长的时间才能完成。 这是因为它会执行一个检查点，并且I/O所需的检查点会被分散在一个显著的时间段， 默认情况下，使用一半你的相互检查点间隔（参见配置参数 checkpoint_completion_target）。 这是你想要的，因为它最大限度地减少对查询处理的影响。 如果你想尽快开始备份，使用：

   ```
   SELECT pg_start_backup('label', true);
   ```

   这强制检查点尽快完成。

3. 执行备份，使用任何方便的文件系统工具，比如tar或者cpio （而不是pg_dump或者pg_dumpall）。 这些操作过程中既不需要关闭数据库，也不需要关闭数据库的操作。

4. 再次以数据库超级用户身份连接数据库，然后发出命令：

   ```
   SELECT pg_stop_backup();
   ```

   这将中止备份模式并自动切换到下一个WAL段。 自动切换是为了在备份间隔中写入的最后一个WAL 段文件可以立即为下次备份作好准备。

5. 只要在备份过程中使用的WAL段文件备份完毕，你的备份工作就完成了。 通过 `pg_stop_backup` 的结果标识的文件是需要形成一套完整备份文件的最后一段。 如果 `archive_mode` 已启用， `pg_stop_backup` 不返回，直到最后段被归档。 这些文件的归档是自动发生的，因为已配置 `archive_command` 。 在大多数情况下，这将迅速发生，但建议您监控您的存档系统以确保没有延迟。 如果归档进程已经落后，因为存档命令失败，它会继续重试直到存档成功，备份完成。 如果您希望在执行 `pg_stop_backup` 时有时间限制，则设定适当的 `statement_timeout` 值。

一些文件系统备份工具发出警告或错误， 如果这些文件他们正试图复制副本处理的变化。 当执行活动数据库的基础备份，这种情况是正常的并且不发生错误。 但是，你需要确保你能辨别这种从实际错误中的投诉。例如，rsync某些版本 返回一个"消失源文件"单独的退出代码，你可以写一个驱动程序脚本接受这个退出代码作为一个非错误情况。 此外，当tar复制它，如

果一个文件被截断时， GNU tar某些版本返回一个错误代码区分致命的错误， 幸运的是，如果文件在备份过程中改变，GNU tar版本1.16和 后期版本退出1，其他错误则返回2。对于GNU tar版本1.23版和 后期版本，您可以使用警告选项的 `--warning=no-file-changed --warning=no-file-removed` 隐藏相关的警告信息。

要保证你的备份转储包括所有数据库集群目录里的文件 (比如 `/usr/local/pgsql/data`)。如果你在使用并未放置在这个目录里的表空间， 也要小心地包含它们，并且要确保你的备份转储归档符号连接是符号连接， 否则，恢复会把你的表空间搞乱。

不过，你可以在备份转储文件里省略集群目录下的 `pg_xlog/` 子目录。 这个略微复杂些的动作是值得的，因为它减少了恢复时候的错误。 如果 `pg_xlog/` 是一个指向集群目录之外的符号连接， 那么这件事情很容易处理，出于性能考虑的时候经常这么做。 你可能还想排除 `postmaster.pid` 和 `postmaster.opts` ， 记录有关运行postmaster的信息，而不谈postmaster。 它最终将使用这个备份。（这些文件可以混淆pg_ctl）。

还有一件事值得一提，那就是 `pg_start_backup` 函数在数据库集群目录里创建了 一个叫 `backup_label` 的文件，它被 `pg_stop_backup` 删除。 这个文件当然也会作为备份转储文件的一部分归档。 这个备份标签文件包含你给予 `pg_start_backup` 的标签字符串， 以及 `pg_start_backup` 运行的时刻，以及起始WAL文件的名字。 如果有混淆，那么我们可以看看备份转储文件里面然后 判断转储文件来自那个备份会话。 然而，这些文件不仅仅是你的信息，它的存在和内容对系统的恢复过程的正确操作是关键的。

我们还可以在服务器停止的时候制作一个备份转储。 在这种条件下，很明显你不能使用 `pg_start_backup` 或者 `pg_stop_backup` ， 并且因此你必须靠自己的手段来跟踪备份转储文件都是那些， 以及相关的WAL文件最远走到哪里。 通常使用上面的在线备份步骤更好些。

24.3.4. 从在线备份中恢复

好，最糟糕的事情发生了，现在你需要从备份中恢复。下面是步骤：

1. 停止服务器，如果它还在运行的话。

2. 如果你还有足够的空间，把整个集群数据目录和所有表空间拷贝到一个临时位置， 以防万一你之后还需要它们。请注意这个预防措施要求你在系统里有足够的剩余 空间来现有库的保持两份拷贝。如果你没有足够的空间， 那么你至少需要把集群数据目录的 `pg_xlog` 子目录的内容拷贝到安全的地方， 因为它们可能包含系统宕掉的时候还没有归档的日志。

3. 然后清理掉所有在该集群数据目录里的现存文件， 以及所有你使用的表空间里根目录下的现存文件。

4. 从你的备份转储中恢复数据库文件。 要小心用正确的所有者(数据库系统用户，而不是 `root` !)和权限恢复它们。 如果你使用了表空间，你可能需要核实在 `pg_tblspc/` 里的符号连接都得到正确恢复。

5. 删除任何目前还在 `pg_xlog/` 里的文件；这些文件来自备份转储， 因此它们可能比目前的老。如果你就根本没有归档 `pg_xlog/` ， 那么重建之， 它具有合适的权限，如果你已经像从前那样建立了，小心确保你重新建立它作为符号链接，

6. 如果你有在步骤 2 里面保存的WAL段文件，那么把它们拷贝到 `pg_xlog/` 。 最好是拷贝它们，而不是把它们移动回来，这样即使发生了糟糕的事情，你需要重启的时候， 你也依然拥有未修改的文件。

7. 在集群数据目录里创建一个恢复命令文件 `recovery.conf` (参阅Chapter 26)。 你可能还需要临时修改 `pg_hba.conf` 以避免普通用户连接， 直到你确信恢复已经正常了为止。

8. 启动服务器。服务器将进入恢复模式并且继续读取它需要的 WAL 归档文件。 在遇见外部错误的应当中止恢复过程，然后重新启动服务器， 这样它会自动继续进行恢复工作。在恢复过程完成后， 服务器将把 `recovery.conf` 改名为 `recovery.done` 以避免不小心因后面的崩溃再次进入恢复模式， 然后开始正常的数据库操作。

9. 检查数据库的内容以确保你已经恢复到你期望的位置。 如果还没有， 回到步骤1。如果全部正常，则恢复 `pg_hba.conf` 成正常状态以允许普通用户登录。

所有这些操作的关键是设置一个恢复命令文件， 这个文件描述你希望如何恢复以及恢复应该走到哪里。 你可以使用 `recovery.conf.sample` (通常安装在安装目录的 `share/` 子目录里)作为原型。你必须在 `recovery.conf` 里面声明的一个东西是 `restore_command` ， 它告诉系统如何拿回归档的WAL文件段。类似 `archive_command` ， 这个是一个脚本命令字符串。它可以包含 `%f` ， 这个变量会被需要的日志文件名替换， 以及 `%p` ， 它会被要拷贝去的日志文件的绝对路径代替。 如果需要在命令里替换真正的 `%` 字符，那么就双写(`%%`)。 最简单的有用命令是类似下面的东西：

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

这个命令将把以前归档的WAL段从 `/mnt/server/archivedir` 目录拷贝过来。 你当然可以使用某些更复杂的东西，甚至是一个要求操作者挂载合适的磁带的shell脚本。

重要的一点是：该命令在失败的时候返回非零值。 如果日志文件没有出现在规档中， 那么该系统将要询问该命令；在问到的时候，它必须返回非零。这个不是错误条件。 并不是所有需要的文件都是WAL分段文件；你应该希望请求 `.backup` 或者 `.history` 的后缀文件。 还要注意 `%p` 路径的基础名将和 `%f` 不一样；不要认为它们是可以互换的。

在归档中没有的WAL分段将在 `pg_xlog/` 中； 这样就允许使用最近没有归档的段。但是在归档中的段将比 `pg_xlog/` 中的优先。

通常，恢复将处理所有可用的 WAL 段， 因此将把数据库恢复到当前时间(或者是在所给出的可用 WAL 段数的情况下， 我们能走到的最近的地方)。 因此，一个正常恢复以"文件没找到"信息结束， 错误信息的确切文本依赖于 `restore_command` 你的选择。你也可能在类似

于 `00000001.history` 文件恢复开头看到错误信息。 这也是正常的，并不表明简单恢复情况下的问题。参阅Section 24.3.5 获取更多信息。

但是如果你想恢复到某些以前的时刻点(比如， 在菜鸟DBA删除你的主要事务表之前)， 那么只需要在 `recovery.conf` 里声明要求的停止点。 你可以通过日期/时间来声明，也可以通过特定事务ID的结束来声明这个停止点， 我们叫做"恢复目标"。目前，只有日期/时间和称为恢复点选项比较有用， 因为我们没有工具来帮助你精确地标识应该使用哪个事务ID。

> **Note:** 请注意停止点必须在备份的终止时间之后(也就是 `pg_stop_backup` 的时间)。 你无法使用一个基础备份恢复到备份正在进行中的某个时刻。要想恢复到该时刻， 你必须回到你以前的基础备份，然后从那个位置向前滚动。

如果在恢复过程中发现在 WAL 数据中存在错误，那么恢复将在错误的地方停止， 并且不会启动服务器。在这种情况下，可以指定一个位于错误点之前的"恢复目标"， 然后从起始点开始重新运行恢复进程，这样恢复就可以正常完成。 如果由于外部原因(系统崩溃、无法读取 WAL 归档)导致恢复失败， 那么可以简单的重新启动恢复过程即可，它将从上次失败的地方继续。 重新启动恢复过程与检查点的操作非常类似： 服务器周期性的强制将其状态记录到磁盘上并更新 `pg_control` 文件以标识已经处理的 WAL 数据不需要被再次扫描。

24.3.5. 时间线

能够把数据库恢复到以前的某个时间点的能力导致了一些类似科幻小说里的时间跟踪 和并行宇宙这样的复杂情况。在数据库最初的历史里， 可能你在周二下午5:15删除掉了一个非常关键的表。但是没有意识到你自己的错误直到周三中午。 有条不紊地拿出备份，恢复到周二晚上5:14的即时备份。在这个数据库宇宙的历史里， 你从来没有删除过那个表。但是假如你后来认识到这么干是错误的， 并且想回到最初的历史中的稍后的点。你没法这么干，因为在数据库运行的时候， 它覆盖了一些WAL段文件的序列，这些序列就在你希望回去的区间里。 因此你的确需要区分在你从那些原始数据库历史生成的WAL中完成即时恢复之 后生成的WAL序列。

为了处理这些问题，PostgreSQL有个叫时间线的概念。 当完成一个归档恢复时，那么就创建一个新的时间线，以表示在该次恢复之后生成的 WAL 记录序列。 时间线ID号是WAL段文件名的一部分， 因此新的时间线并不会覆盖以前的时间线生成的 WAL 数据。 实际上我们可以归档许多不同的时间线。虽然这些看起来像没用的特性， 但它却可能常常是救命稻草。考虑一下你并不很确信应该恢复到那个时刻的情况， 这个时候你不得不做好几次试验性即时恢复然后从中找到旧历史中最好的分支。 如果没有时间线，那么这个过程可能很快就会导致无法管理的混乱。 有了时间线，你可以恢复到任意以前的状态， 包括恢复到你后来放弃的时间线分支的状态。

每当创建一个新的时间线的时候，PostgreSQL都创建一个"时间线历史"文件， 它显示自己从哪个时间线分出来，以及何时分出来的。 这些历史文件是在从包含多个时间线的归档中进行恢复时， 允许系统选取正确 WAL 段文件的必要文件。因此， 它们像 WAL 段文件一样归档到

WAL 归档里。 历史文件只是很小的文本文件(不像段文件很大)， 所以独立地保存他们代价很小，也值得做。如果你喜欢， 你可以在历史文件里加入注释，记录自己为什么设置这个时间线以及如何设置的等信息。 这样的注释会在你有厚厚一堆不同的时间线需要选择和分析的时候特别有价值。

恢复的缺省的行为是沿着与基础备份的同一个时间线恢复。 如果你想恢复到某些子时间线，也就是，你想回到某些本身就是在开始恢复之后发生的状态。 你需要在 `recovery.conf` 里声明目标时间线ID。 你无法恢复到比基础备份更早的时间线分支。

24.3.6. 技巧和例子

给出了配置连续归档的一些技巧。

24.3.6.1. 单机热备

可以使用PostgreSQL备份工具产生单机热备。这些都是不能使用时间点恢复的备份， 往往备份和恢复比pg_dump转储速度更快。 （他们也比pg_dump转储更大，因此在某些情况下速度优势可能是否定的）。

作为基础备份，最简单的方式是使用pg_basebackup工具产生单机热备份。 当调用它时，如果你有 `-X` 参数，那么所有需要使用备份的事务日志将包含在自动备份中， 并且不需要特殊操作恢复备份。

如果在复制备份文件中需要更大的灵活性，较低的水平过程可更好用于单机热备份。 为了准备低水平单机热备份，设置 `wal_level` 到 `archive` (或者 `hot_standby`), `archive_mode` 为 `on` ， 并且只有当开关文件存在时， 建立 `archive_command` 执行存档。例如：

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgs
```

当 `/var/lib/pgsql/backup_in_progress` 存在时，这个命令将执行归档。 否则默默返回零退出状态（允许PostgreSQL回收不需要的WAL文件）。

有了这个准备，备份可以使用如下脚本：

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

开关文件 `/var/lib/pgsql/backup_in_progress` 是第一次创建， 使已完成的WAL文件的归档发生。在备份后删除开关文件。归档的WAL文件然后添加至备份， 使两个基础备份和所有必需的WAL文件是相同的tar文件的一部分。 请记住，错误处理添加到您的备份脚本中。

24.3.6.2. 压缩归档日志

如果归档存储大小是一个问题，你可以使用gzip 压缩归档文件：

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

在恢复过程中你需要使用gunzip：

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

24.3.6.3. `archive_command` 脚本

许多人选择使用脚本定义他们的 `archive_command` ， 因此 `postgresql.conf` 记录看起来很简单：

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

任何你想在归档过程中使用不只是独立命令时。 使用一个单独的脚本文件是可取的，这允许所有的复杂性在脚本中管理， 这可以使用流行的脚本语言书写，如bash或者perl。

可能在脚本中解决的所需例子包含：

- 拷贝数据到安全异地数据存储
- 计量WAL文件以致于他们每三个小时改变，而不是一个时间。
- 其他备份和恢复软件接口
- 监控软件报告错误接口

> **Tip:** 当使用 `archive_command` 脚本时，期望启动logging_collector。 来自脚本写入stderr中的任何消息将出现在数据库服务器日志中， 如果失败，则允许简单地诊断复杂配置。

24.3.7. 警告

目前，在线备份技术还有几个局限。它们可能在将来的版本中修补：

- 在Hash索引上的操作目前没有使用WAL记录日志，所以重放就不会更新这些索引类型。这将意味着任何新的插入被索引忽略，已更新行显然会消失，并且已删除行将仍然保留指针。 换句话说，如果你修改了带有hash索引的表，那么你将获得备用服务器上不正确的查询结果。 当完成恢复时，建议是在完成恢复操作之后手工REINDEX每个这样的索引。
- 如果在进行数据库备份的时候发出一个CREATE DATABASE命令， 然后在这个过程中 `CREATE DATABASE` 命令拷贝的模板数据库被修改了， 那么用这个备份进行恢复的数据库很有可能导致这些修改也传播到新创建的数据库中去。 这个行为当然是不愿意看到的。 为了避免这个风险， 最好在进行数据库备份的时候不要修改任何模板数据库。
- CREATE TABLESPACE命令是用文本的绝对路径记录WAL日志的， 因此会以相同的绝对路径重新创建。如果日志是在另外一台机器上重放， 那么这个行为可能不是我们想要的。即使在同一台机器， 但是在一个新的数据目录里重放日志，都很可能是危险的： 重放仍将会覆盖原来的表空间的内容。 为了避免这类的潜在问题， 最好的方法是在创建或者删除表空间之后进行一次新的基础备份。

还要注意，缺省的WAL格式体积相当大，因为它包含许多磁盘页快照。 这些磁盘页快照是设计来支持崩溃恢复的， 因为我们可能需要修补部分写入的磁盘页。根据你的系统硬件和软件的不同， 这种部分写入的危险可能是微乎其微的。这种情况下， 你可以通过使用full_page_writes关闭磁盘页面快照， 从而大大减少归档日志的总尺寸(在你这么做之前，阅读Chapter 29里面的注意和警告)。 关闭页面快照并不阻止日志使用PITR操作。 一个将来需要开发的功能是在 `full_page_writes` 打开的时候， 通过删除不需要的磁盘页拷贝来压缩归档的 WAL 数据。同时， 管理员可以通过尽量合理地增加检查点的时间间隔来减少包含在WAL里的页面快照。

Chapter 25. 高可用性与负载均衡，复制

Table of Contents

多个数据库服务器可以协同工作， 比如在主服务器失效的时候备份服务器立即取代它的位置(高可用性)， 或者几台机器同时服务于同一个数据库(负载均衡)。 理想状态多台服务器之间可以无缝协作。 为静态页面提供服务的Web服务器可以轻松的通过将web请求分摊到多台机 器从而实现负载均衡。 事实上，只读数据库也能轻松的以相同的方法实现负载均衡。 不幸的是，大多数数据库服务器都需要同时处理混合的读/写请求， 将这些数据库联合起来工作是件很麻烦的事。 虽然只读数据只需要在每台服务器上复制一份即可， 但是在任何一台服务器上的写动作都必须传播到其它所有服务器上， 这样才能保证将来对这些已修改数据的读取返回一致的结果。

这个写同步问题就是导致多台服务器协同工作麻烦重重的最基本原因。 有多种解决此问题的方法，其思路也各不相同， 但都不是既简单又高效的方案。

有一种解决方案是仅允许单独的一台"主"服务器修改数据， 可以修改数据的服务器称为只读/写，*master*或者*primary*服务器。 跟踪“主”服务器数据变化的叫备 或者从服务器。 备用服务器不能连接到主服务器，直到它晋升为热备服务器。 可以接受连接而且只读服务器称为热备服务器。

一些方案是"同步的"， 意思是直到所有服务器都完成了某个修改数据的事务之后， 该事务才被认为是已经完成的。 这将确保失效切换不会丢失任何数据并且所有服务器都将返回一致的结果。 另一些方案是"异步的"，这种方案允许在事务提交之后 与传播到所有其它服务器之间有一小段延时， 但是在切换到备份服务器的时候某些事务可能会丢失， 并且不同的服务器可能返回不一致的结果。 当同步可能会很慢的时候可以使用异步通信。

还可以按照粒度对解决方案进行分类。 某些方案只能将整个数据库集群作为一个整体， 而某些方案可以针对每个数据库或每张表分别做不同的处理。

在选择任何失效切换或负载均衡方案的时候都必须考虑性能因素。 功能和性能不可兼得，比如，一个完全同步的解决方案在慢速网络上可能削减性能一半以上， 而完全异步的方案可能仅对性能有极其微小的影响。

下面的部分大致描述了各种常见的失效切换、复制、 负载均衡方案。 glossary 也是可用的。

25.1. 不同解决方案的比较

共享磁盘失效切换

共享磁盘失效切换通过仅保存一份数据库副本来避免花在同步上的开销。 这个方案让多台服务器共享使用一个单独的磁盘阵列。 如果主服务器失效， 备份服务器将立即挂载该数据库，就像是从一次崩溃中恢复一样。这个方案允许快速的失效切换并且不会丢失数据。

共享硬件的功能通常由网络存储设备提供， 也可以使用完全符合POSIX行为的网络文件系统(参阅Section 17.2.1)。 这种方案的局限性在于如果共享的磁盘阵列损坏了， 那么整个系统将会瘫痪。 另一个局限是备份服务器在主服务器正常运行的时候不能访问共享的存储器。

文件系统复制（块设备）

一种改进的方案是文件系统复制：对文件系统的任何更改都将镜像到备份服务器上。 这个方案的唯一局限是必须确保备份服务器的镜像与主服务器完全一致— 特别是写入顺序必须完全相同。DRBD是Linux上的一种流行的文件系统复制方案。

事务日志传送

热备份服务器可以通过读取WAL记录流来保持数据库的当前状态。 如果主服务器失效，那么热备份服务器将包含几乎所有主服务器的数据， 并可以迅速的将自己切换为主服务器。这是一个异步方案， 并且只能在整个数据库服务器上实施。

使用基于文件的日志传送或流复制，或两者相结合。 前者参阅Section 25.2, 后者参阅Section 25.2.5。 请参阅Section 25.5获取关于热备的信息。

基于触发器的主备复制

这个方案将所有修改数据的请求发送到主服务器。 主服务器异步向从服务器发送数据的更改信息。 从服务器在主服务器运行的情况下只应答读请求。对于数据仓库的请求来说， 从服务器非常理想的。

Slony-I是这个方案的一个例子，它支持针对每个表的粒度并支持多个从服务器。 因为它异步、批量的更新从服务器， 在失效切换的时候可能会有数据丢失。

基于语句的复制中间件

可以使用一个基于语句的复制中间件程序截取每一个SQL查询， 并将其发送到某一个或者全部服务器。每一个服务器都独立运行。 读-写请求发送给所有服务器，所以每个服务器接收到任何变化。但是只 读请求则仅发送给某一个服务器，从而实现读取的负载均衡。

如果只是简单的广播修改数据的SQL语句， 那么类似 `random()`, `CURRENT_TIMESTAMP` 以及序列函数在不同的服务器上将生成不同的结果。 这是因为每个服务器都独立运行并且广播的是SQL语句而不是如何对行进行修改。 如果这种结果是不可接受的，那么中间件或者应用程序

必须保证始终从同 一个服务器读取这些值并将其应用到写入请求中。 另外还必须保证每一个事务必须在所有服务器上全部提交成功或者全部回滚， 或者使用两阶段提交(PREPARE TRANSACTION 和COMMIT PREPARED)。 Pgpool-II和Continuent Tungsten是这种方案的实例。

异步多主服务器复制

对于那些不规则连接的服务器(比如笔记本电脑或远程服务器)， 要在它们之间保持数据一致是很麻烦的。 在这个方案中，每台服务器都独立工作并周期性的与其他服务器通信以识别相互冲突的事务。 可以通过用户或者冲突判决规则处理出现的冲突。

同步多主服务器复制

在这种方案中，每个服务器都可以接受写入请求， 修改的数据将在事务被提交之前必须从原始服务器广播到所有其它服务器。 过多的写入动作将导致过多的锁定，从而导致性能低下。 事实上，在多台服务器上同时写的性能总是比在单独一台服务器上写的性能低。 读请求将被均衡的分散到每台单独的服务器。 某些实现使用共享磁盘来减少通信开销。 同步多主服务器复制方案最适合于读取远多于写入的场合。 它的优势是每台服务器都能接受写请求—因此不需要在主从服务器之间划分工作负荷。 因为在服务器之间发送的是数据的变化， 所以不会对非确定性函数(比如 `random()`)造成不良影响。

PostgreSQL不提供这种类型的复制。 但是PostgreSQL的两阶段提交(PREPARE TRANSACTION和 COMMIT PREPARED) 可以用于在应用层或中间件代码中实现这个功能。

商业解决方案

因为PostgreSQL是开放源代码并且很容易被扩展， 许多公司在PostgreSQL的基础上创建了商业的闭源解决方案， 提供独特的失效切换、复制、 负载均衡功能。

Table 25-1总结了以上所列的各种解决方案的能力。

Table 25-1. High Availability, Load Balancing, and Replication Feature Matrix

Feature	Shared Disk Failover	File System Replication	Transaction Log Shipping	Trigger-Based Master-Standby Replication	Statemen Based Replicati Middlewa
Most Common Implementation	NAS	DRBD	Streaming Repl.	Slony	pgpool-II
Communication Method	shared disk	disk blocks	WAL	table rows	SQL
No special hardware required	•	•	•	•	•
Allows multiple master servers	•	•	•		
No master server overhead	•	•	•		
No waiting for multiple servers	•	with sync off	•	•	
Master failure will never lose data	•	•	with sync on	•	•
Standby accept read-only queries	with hot	•	•	•	•
Per-table granularity	•	•	•		
No conflict resolution necessary	•	•	•	•	•

有几个解决方案不适合上边这些分类：

数据分区

数据分区将表拆分为数据集。每个数据集只有一台服务器可以修改。 例如，数据可以按办事处进行分区，例如， 伦敦和巴黎，每个办公室用一个服务器。 如果查询需要伦敦和巴黎相结合的数据，应用程序可以查询两台服务器， 或主/备用复制可以用来保持每个服务器上有其他办公室的只读数据副本。

多服务器并行查询执行

许多上述解决方案允许多个服务器来处理多个查询， 但不是允许单个查询使用多个服务器来更快完成。 此解决方案允许多个服务器上单个查询同时运行。 它通常被通过服务器之间的数据分开而执行其查询的一部分， 并将结果返回到中央服务器，由它来联合结果并返回给用户。 Pgpool-II有这种能力。 也可以使用PL/Proxy工具集实现。

25.2. 日志传送备份服务器

连续归档可以配合随时准备取代失效主服务器的一个或多个备份服务器， 用于创建一个高可用性(HA)集群。 这个能力通常被称为热备份或日志传送。

虽然主服务器和备份服务器只是松散的耦合在一起， 但它们必须同时运行。主服务器以连续归档模式运行， 备份服务器以连续恢复模式运行并从主服务器不停的读取WAL文件。 因为数据库的表无需为此进行任何改变，所以与其它复制方法相比， 额外的管理开销很小。并且这种方法对主服务器的性能影响也很小。

直接从一个数据库服务器移动WAL到另一个服务器通常被称为日志传送(LogShipping)。PostgreSQL实现了基于文件的日志传送， 意思是WAL记录每次移动一个完整的文件(WAL段)。 WAL文件（16MB）可以被轻易的在任意两个地点之间传送， 不管是与邻近的系统还是地球另一面的系统。 所需带宽取决于主服务器的事务发生速度。 基于记录的日志传送更加细粒度，并且WAL流在网络连接中增量改变。

日志传送是异步的，也就是WAL记录在事务提交之后才被传送。 也就是说主服务器遭遇致命故障后尚未传送的事务数据将会丢失。 数据丢失的长度可以使用`archive_timeout`加以限制，比如限制为几秒钟。 当然这么小的设置也导致了传送带宽的大幅增长。 流复制（参阅Section 25.2.5）允许数据丢失的更小窗口。

恢复性能足够好，备份服务器一旦被激活通常只有很短的时间不能使用。 因此，我们认为这个方案可以作为热备份来提供高可用性。 将服务器从一个已归档的基础备份中恢复将可能耗费大量时间， 所以这个方案只能用于灾难恢复而不能用于提供高可用性。 备用服务器还可以用于只读查询，在这种情况下它被称为热备份服务器。 参见Section 25.5获取更多信息。

25.2.1. 规划

至少从数据库服务器的角度看， 创建主服务器和备份服务器并令两者尽可能完全相同是非常明智的。 特别是表空间的路径名必须保持完全一致， 这样主服务器和备份服务器就必须拥有同样的表空间挂载路径(如果使用了表空间的话)。 需要记住的是如果在主服务器上执行了CREATE TABLESPACE命令， 那么该命令需要的任何新挂载点必须在执行该命令之前同时在主服务器和备份服务器 上创建。 硬件不必完全相同，但是经验显示维护两个完全相同的系统比维护两 个不同的系统要少许多麻烦。 无论如何，应尽量保持体系结构相同— 比如一个是32-bit系统另一个是64-bit系统将不能正常工作。

通常，在主PostgreSQL版本不同的服务器之间传送日志是不可能的， 它是PostgreSQL全球开发组在次要版本升级中不能改变磁盘格式的一种策略。 在主服务器和备份服务器上运行不同的次要版本可能成功。 但是，没有正式支持，建议你尽可能的保持主服务器和备用服务器

在同一个级别上。 在进行版本升级的时候，正确的做法是首先升级备份服务器— 因为新版本的服务器通常可以读取老版本的WAL文件，但反之则不然。

25.2.2. 备用服务器操作

在备用模式，该服务器连续应用从主服务器收取的WAL。 备服务器可以从一个WAL归档（参阅restore_command） 或直接通过一个TCP连接（流复制）从主服务器上读取WAL。 备服务器也可以在备用集群 `pg_xlog` 尝试查找恢复任何WAL。 这通常发生在服务器重启后，当备服务器重播，在备服务器重启前， 从主服务器流复制的WAL，但是你也可以手工复制文件到 `pg_xlog` ， 在任何时候可以重播它们。

在启动，备服务器恢复可用在所有的WAL开始存档位置,调用 `restore_command` 。 一旦它到达可用WAL的结束， `restore_command` 失败， 将尝试恢复 `pg_xlog` 目录下任何可用的WAL。 如果那也失败了，并且已经配置了流复制，则尝试连接到主服务器， 从在归档或 `pg_xlog` 找到最后一条有效的记录开始WAL流。如果那也失败了， 或没有配置流复制，或连接断开，备服务器再次回到步骤1，循环尝试从归档里恢复文件。 从归档， `pg_xlog` ，通过连续流复制直到服务器停止或有触发器文件触发的失效切换时。

当运行 `pg_ctl promote` 时，或者找到一个触发文件（ `trigger_file` ）时， 退出备用模式并且服务器切换到正常运行。 在失效切换前，将立即恢复归档或 `pg_xlog` 任何可用的WAL， 但不做尝试连接主服务器。

25.2.3. 为备用服务器准备主服务器

在主服务器上设置连续归档到一个备服务器可访问的存档目录， 正如Section 24.3所描述的。 即使主服务器关掉， 从备服务器应该可以访问这个归档位置。 即它应该驻留在被服务器自身或其它可信赖的服务器， 而不是主服务器。

如果你想使用流复制，在主服务器上设置认证，允许从备用服务器复制连接；在 `pg_hba.conf` 提供一个或多个合适项使用数据库字段设置 `replication` 。 还要在主服务器的配置文件确保设置 `max_wal_senders` 足够大。

启动备用服务器做一个基准备份，参见Section 24.3.2。

25.2.4. 建立备用服务器

要建立备用服务器，从主服务器恢复基准备份（参阅Section 24.3.4）。 在备用服务器的集群数据目录，创建一个恢复命令文件 `recovery.conf` , 开启 `standby_mode` 。 设置 `restore_command` 为一条从WAL归档复制文件的简单命令。如果为了高可用性目的计划有多个备用服务器， 设置 `recovery_target_timeline` 为 `latest` ， 使得备用服务器按照发生故障转移到另一个备用服务器的时间变化。

> **Note:** 不要使用内置在这里描述的备用模式pg_standby或类似的工具。 如果该文件不存在， `restore_command` 应该立即返回。 如果必要服务器将再次尝试这个命令。 关于使用工具像pg_standby的详情参阅Section 25.4。

如果你想使用流复制，在 `primary_conninfo` 填写一个libpq连接串， 其包括主机名（或IP地址）和连接到主服务器需要的其它详细信息。 如果主服务器需要个密码验证，也要在 `primary_conninfo` 指定所需要的密码。

如果你要建立高可用目的备服务器，设置WAL归档， 像主服务器的连接和身份验证，因为在失效切换后， 备服务器要作为主服务器运行。

如果你使用WAL归档，其大小可以使用archive_cleanup_command这个参数设置最小， 用来删除那些备服务器不再需要的文件。 专门设计的pg_archivecleanup这个实用程序就是在通常的单备配置里， 使用 `archive_cleanup_command` 的。参阅pg_archivecleanup。 请注意，如果你使用备份目的归档， 你仍要保留需要恢复的至少最新的基准备份文件， 即使备服务器不再需要。

`recovery.conf` 的一个简单例子：

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

你可能有任何数目的备服务器，但是如果你用流复制， 确保你在主服务器上设置的 `max_wal_senders` 足够大允许它们同时连接。

25.2.5. 流复制

与基于文件日志传送相比，流复制允许保持备服务器更新。 备服务器连接主服务器，其产生的流WAL记录到备服务器， 而不需要等待填写WAL文件。

流复制是异步的，参阅Section 25.2.7， 在主服务器上提交事务和备用服务器上变化可见之间有一个小的延迟。 这个延迟远小于基于文件日志传送，通常1秒内足够与负载保持。 使用流复制，为减少数据丢失窗口 `archive_timeout` 不是必要的。

如果使用流复制而不是基于文件连续归档， 你要在主服务器设置 `wal_keep_segments` 为一个足够大的值以使不太早的回收旧WAL段， 当备服务器可能仍需要它们赶上。如果备服务器落后太多， 需要用一个新基准备份重新初始化。如果你设置一个备服务器可访问的WAL归档， `wal_keep_segments` 是不必要的， 作为备服务器总是使用归档来赶上。

要使用流复制，建立一个基于文件的日志传送备服务器描述在Section 25.2。 该步将一个基于文件的日志传送备服务器转为流复制备服务器， 在 `recovery.conf` 文件中设置 `primary_conninfo` 指向主服务器。 在主服务器上设置listen_addresses和身份验证选项

（参阅 `pg_hba.conf` ）， 因此备用服务器可以连接到在主服务器的 `replication` 伪数据库（参阅Section 25.2.5.1）。

系统上支持保持活动的套接字选项， 设置tcp_keepalives_idle, tcp_keepalives_interval和tcp_keepalives_count 帮助主机及时发现断开的连接。

设置备用服务器的最大并发连接数。 （参阅max_wal_senders获取更多详细信息）。

当启动了备服务器并且正确设置了 `primary_conninfo` , 该备服务器在回放所有可用的WAL文件后，将连接到主服务器。如果成功建立了该连接， 你将在备服务器中看到WAL接收进程，并且在主服务器相应的一个WAL发送进程。

25.2.5.1. 身份验证

复制的访问权限设置是很重要的，所以只有受信任的用户可以读取WAL流， 因为很容易从中提取权限信息。备服务器必须验证作为主服务器的超级用户或者有 `REPLICATION` 权限的用户。 建议为复制创建一个带有 `REPLICATION` 和 `LOGIN` 权限的专有用户账号。 当 `REPLICATION` 权限有很高权限时，不允许用户修改主服务器上的任何数据， 其中 `SUPERUSER` 就是这样的。

由一条 `pg_hba.conf` 记录指定 `replication` 在 `_database_` 字段， 控制客户端的复制验证。例如，如果备服务器是运行在主机IP `192.168.1.100` 和复制时超级用户名为 `foo` ， 管理员可以在主服务器 `pg_hba.conf` 文件里添加下面行：

```
# Allow the user "foo" from host 192.168.1.100 to connect to the primary
# as a replication standby if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

主服务器的主机名和端口号，连接用户名，和在 `recovery.conf` 文件指定的密码。 该密码也可以在备服务器的 `~/.pgpass` 文件里设置。 （在 `_database_` 字段指定 `replication` ）。例如， 如果主服务器是运行的主机IP `192.168.1.50` ,端口号 `5432` , 复制时用户名为 `foo` ， 和密码为 `foopass` ， 管理员可以在备服务的 `recovery.conf` 文件里添加下面行：

```
# The standby connects to the primary that is running on host 192.168.1.50
# and port 5432 as the user "foo" whose password is "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. 监控

流复制的一个重要的健康指标是在主服务器生成的WAL记录数， 而不是在备服务器应用的数量。通过比较在主服务器当前WAL写的位置和备服务器收到 的最后一个WAL位置，就可以计算出这种滞后。 在主服务器上使用 `pg_current_xlog_location` 和在备服务器上使用

`pg_last_xlog_receive_location` 可以分别检索到它们（参阅Table 9-60和 Table 9-61关于详细信息）。 在备服务器收到最后的WAL位置也会进程状态的WAL接收进程显示， 使用 `ps` 命令显示（参阅Section 27.1关于详细信息）。

你可以通过 `pg_stat_replication` 视图回收WAL发送程序列表。 在 `pg_current_xlog_location` 和 `sent_location` 字段之间的不同可能表明 主服务器在大负载下，而备库上 `sent_location` 和 `pg_last_xlog_receive_location` 的不同可能表明网络延迟，或者备库也处于重大负载之下。

25.2.6. Cascading Replication

级联复制功能允许备用服务器接受复制连接和到其他备库的流WAL记录，充当延迟。 这可以用来降低直接连接主库的数量，同时也尽量减少站点间的带宽开销。

同时充当接收器和发送器的待机称为级联 待机状态。更直接地连接到主库的备库作为上游服务器已知， 而备用服务器较远的下游服务器。级联复制不放在数量限制或 下游服务器的安排， 虽然每个备库连接到一个上流服务器， 而最终链接到一个单一的主/主服务器。

级联备库不仅仅发送来自主库的WAL记录，而且来自归档的记录。 因此即使一些上流连接的复制连接终止， 连接复制继续往下只要有新的WAL记录可用。

级联复制当前是异步的，同步复制（参阅Section 25.2.7）设置当前不影响级联复制。

热备用反馈传播到上游，无论级联配置。

如果上游备用服务器上升成为新主库，下游服务器将继续流向新主库， 如果 `recovery_target_timeline` 设置为 `'latest'` 。

使用级联复制，建立连锁备库从而接受复制连接（即设置 max_wal_senders和hot_standby，以及配置host-based authentication）。 在下游备库指向级联备库中你还需要设定 `primary_conninfo` 。

25.2.7. 同步复制

PostgreSQL流复制缺省是异步的。如果主服务器崩溃，然后一些事务承诺不得不 复制到备用服务器，造成数据丢失。数据丢失量是与故障转移时的复制延迟是成比例的。

同步复制提供确认所有变化已经由事务转移到一个同步备用服务器的能力。 这延伸一个事务提交的耐久性标准。 这种级别的保护是指2-安全复制的计算机科学理论。

当请求同步复制，每次提交的写事务将等待直到确认收到提交已被写入到主库和备用服务器磁盘上的事务日志。 这些数据丢失的可能性是如果主库和备库同一时间遭受崩溃。 这可以提供一个更高层次的耐久性，但如果系统管理员对两个服务器的安置和管理非常谨慎。 等待确

认增加用户的信心，在服务器崩溃的情况下更改将不会丢失，而是增加请求事务的响应时间。 最小等待时间是主库到备库的往返时间。

只读事务和事务回滚不需要等待从备用服务器的回复。 子事务提交不等待从备用服务器的响应，只有顶级提交。 长时间运行动作如加载数据或索引建立不等待直到最后提交信息。 所有的两阶段提交的行动需要提交等待，包括准备和提交。

25.2.7.1. 基础配置

一旦流复制已配置，配置同步复制仅需要一个额外的配置步骤：synchronous_standby_names必须设置为 一个非空值。 `synchronous_commit` 也必须设置为 `on` ， 但因为这是默认值，通常是没有改变的。 （参见Section 18.5.1和 Section 18.6.2）。 这种配置将导致每次提交会等待确认备库书面提交记录到持久存储。 `synchronous_commit` 可以由个人用户设置， 所以它可以被配置在配置文件中，特别是 用户或数据库中，或动态的应用程序中， 以保证每个事务的基础上控制耐久性。

一个提交记录已被写入到主库磁盘上，WAL记录随后被发送到待机状态。 每次一批新的WAL数据被写入磁盘时待机发送答复消息， 除非 `wal_receiver_status_interval` 在待机状态设置为零。 如果待机是第一个匹配的待机， 在主库上指定 `synchronous_standby_names` ， 从待机状态得到的答复信息将被用于唤醒用户的提交记录已被接收的等待确认。 这些参数允许管理员指定哪些备用服务器应同步备用。 注意，同步的配置复制主要针对主服务器。 命名的备用服务器必须直接连接到主库上;主库并不了解下游待机使用级联复制的服务器。

设置 `synchronous_commit` 到 `remote_write` 将 导致每个提交等待确认备库已经收到提交记录并且写入到自己的操作系统， 但不是为了该数据被刷新到磁盘上的备份。 这个设置提供了耐用性较弱保证对比 `on` ： 在待机状态下可能会失去操作系统崩溃时的数据， 虽然不是PostgreSQL崩溃。 然而，这是在实践中有用的设置，因为它可以减少事务响应时间。 如果主库和备库崩溃，并且主库的数据库同时被破坏了，只能发生数据丢失。

如果要求快速关机，用户将停止等待。然而，作为使用异步复制的时候，服务器将不完全 关闭直到所有WAL记录转移到目前连接的备用服务器。

25.2.7.2. 规划性能

同步复制通常需要仔细规划并且放置备用服务器，确保应用程序执行性能。等待 没有充分利用系统资源，但事务锁继续直到确认转移。作为一个结果， 不小心的使用同步复制会降低数据库应用性能，由于响应时间的增加和更高的竞争。

PostgreSQL允许应用程序开发人员通过复制来指定所需的耐久性水平。这可以 对系统的总体说明， 虽然它也可以被指定为特定用户或连接，甚至个别交易。

例如，一个应用程序的任务可能包括：10%的变化是重要的客户资料，而 90%的变化是不太重要的数据，如果它丢失，企业还可以更好生存，比如用户之间的聊天信息。

在应用水平（主库）指定同步复制选项，我们可以为大多数重要的变化提供同步复制，没有放缓总工作量体积。对于允许同步复制的高性能应用的效益来说， 应用程序级别的选项是一个重要而实用的工具。

你应该考虑到网络带宽必须大于WAL数据生成率。

25.2.7.3. 高可用性规划

当 `synchronous_commit` 设置为 `on` 或者 `remote_write` 将等待直到同步备用响应，则进行提交。 如果最后，或只发生待机崩溃，则响应不会发生。

为避免数据丢失最好的办法是确保你不会失去你最后的同步备份。 这可以通过使用 `synchronous_standby_names` 命名多个 潜在的同步备用实现。 第一个命名的备库将作为同步备库使用。如果第一个失败， 则备用列表将接管同步备用角色。

当一个备库首先依附在主库时，不能正确地同步。 这是 `catchup` 方式所描述的。 一旦备库和主库之间的滞后达到零，第一次我们将实时状态 `流` 。 备库被创建后，持续追赶时间可能长。如果备库关闭， 然后追赶时期将随着待机时间长度而增加。 一旦已经达到 `流` 状态，备库是唯一能够成为同步备用。

如果主库启动而提交等待确认，这些等待事务将被标记为完全提交，一旦主数据库恢复。 没有办法确保在主库崩溃的时候所有备库收到所有优秀的WAL数据。 一些事务可能不会在待机时显示提交，即使他们表现为主库已提交。 我们提供的保障是应用程序将不会接收事务成功提交的明确承认， 直到被备库安全接收WAL数据是已知的。

如果你真的失去了你最后的备用服务器，你应该禁用 `synchronous_standby_names` 并且在主服务器 重新加载配置文件。

如果主库是从剩余备用服务器分离的， 你应该故障转移到那些其他剩余备用服务器的最佳人选。

如果你需要重新创建备用服务器，当等待事务时，确保pg_start_backup()和pg_stop_backup()在带有 `synchronous_commit` = `off` 的回话中运行， 否则这些请求将永远等待备库出现。

25.3. 失效切换

如果主服务器失败，则备服务器应该开始失效切换处理。

如果备服务器失败，则没有失效切换需要考虑。如果可以重启备用服务器， 甚至一段时间后，也可以立即重启恢复进程，发挥重启恢复的优势。 如果不能重启备服务器，则应该创建一个全新的备服务器实例。

如果主服务器失败，并且备服务器成为新主服务器，然后旧主服务器重启， 你必须有一个通知旧主服务器，其不再是主服务器的机制。 这有时被称为STONITH（在头去掉其它节点），这是必要的，以避免系统都认为它们是主服务器的情况下， 这将导致混乱和最终数据丢失。

许多失效切换系统只使用两个系统，主备服务器，通过某种心跳机制， 不断验证两者连接和主服务器的活力。 也可以使用一个第三方的系统 （称为"证人服务器"），以防止某些情况下不适当的失效切换， 但额外的复杂性可能是不值得的，除非设置它为充分仔细和严格的测试。

PostgreSQL不提供所需的用来确定主服务器失败， 并通知备用数据库服务器的系统软件。 存在许多这样的工具和成功失效切换所需的集成操作系统的工具，如IP地址迁移。

一旦发生失效切换到备服务器，仅有一台服务器运行。这就是所谓的退化状态。 前者备服务器现在是主服务器，但前者主服务器是可能会停留下来。 要返回正常运行，必须重建一个备服务器，无论是在以前的服务器， 或在第三，可能是新的系统。一旦完成主备服务器，可以考虑转换角色。 有些人选择使用第三方服务器，提供新主服务器的备份直到新备服务器重建， 尽管清楚这个复杂的系统配置和操作流程。

所以从主到备服务器可以快速切换，但需要一些时间重新准备失败切换集群。 定期主备服务器切换是有用的,因为它允许定期停机进行每个系统的维修。 这也是作为一个测试，以确保故失效切换机制，当你需要时会真的工作。 建议写管理操作流程。

要触发日志传送备服务器的失效切换，运行 `pg_ctl promote` 或者创建一个触发文件, 这个文件是通过在 `recovery.conf` 文件中 `trigger_file` 设置指定的文件名和路径。 如果你计划使用 `pg_ctl promote` 进行失效切换，则不需要 `trigger_file` 。 如果你设置报告服务器仅仅用于卸载主服务器的只读查询， 而不是针对高可用性的目的，那么你不需要推动它。

25.4. 日志传送的替代方法

一种替代在前节描述的内建备用模式的方法是使用 `restore_command` 轮询归档位置。 这是只能在8.4及以下版本选择使用。在此设置 `standby_mode` 关闭， 因为你要实现备服务器运行你自己所需的轮询。 请参考pg_standby模块关于这类的实现。

请注意在这种模式，服务器将一次应用一个WAL文件，所以如果你使用备服务器对于查询（见热备）， 在主服务器中的动作和当这个动作在备服务器中可见之间有个延迟， 相应的时间用在填写WAL文件。 `archive_timeout` 可以使延迟较短。 还要注意你不能用这种方法结合流复制。

主备用服务器上发生的操作是正常的连续归档和恢复任务。 两个数据库服务器相联系的一点是两者共享的WAL归档文件： 主写入归档， 备从归档读取。必须小心，以确保从单独的主服务器， 不会混在一起或混淆WAL归档。如果只是备服务器操作要求， 归档需要并不大。

使松散耦合的两个服务器一起工作简直是奇迹，在备服务器上简单使用 `restore_command` ， 当询问下一个WAL文件，等待其为主服务器可用的。 在备服务器的 `recovery.conf` 文件指定 `restore_command` 。 通常恢复进程将从一个WAL归档中请求文件，如果该文件不可用，则报告失败。 对备服务器进程来说下一个WAL文件不可用是正常的，因此备服务器进程需要等待它出现。 对于在 `.backup` 或者 `.history` 文件结束不需要等待， 并且返回一个非零值。等待 `restore_command` 可以写为一个自定义脚本， 即循环轮询下一个WAL文件的存在。还必须有一些方法来触发失效切换， 应该中断的 `restore_command` ，跳出循环， 并返回备用服务器一个文件未找到错误。这两端的恢复和备用服务器， 然后将作为一个正常的服务器。

一个合适 `restore_command` 的伪码是：

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

在pg_standby模块中提供一个等待 `restore_command` 的实际例子。 应该用来作为参考如何正确地贯彻执行上述逻辑。它也可以扩展需要， 以支持特定的配置和环境。

触发失效切换的方法是规划和设计的一个重要组成部分。 一个潜在的选项是 `restore_command` 命令。每个WAL文件执行一次， 但是运行 `restore_command` 的进程对于每个文件创建和消亡的， 所以没有守护进程或服务器进程和信号或不能使用的信号处理。 因此， `restore_command` 不适合触发失效切换。使用简单超时机制可能的， 尤其如果与已知

的 `archive_timeout` 在主服务器上配合设置使用。 尽管，这比较容易出错，因为网络问题或繁忙的主服务器可能有足够的启动失效切换。 如果可以安排，通报机制如显式创建一个触发器文件是理想的。

25.4.1. 实施

配置备用服务器，使用这种替代方法简短步骤如下。对于每一步的细节， 请参阅前面的章节。

1. 建立主备系统尽可能接近相同，包括两个PostgreSQL副本在相同版本级别。
2. 设置从主服务器上连续归档到备服务器WAL归档目录。 确保在主服务器上相应的设置archive_mode, archive_command和archive_timeout。 (参阅Section 24.3.1)。
3. 做一个主服务器的基准备份(参阅Section 24.3.2)， 在备服务器上加载这个数据。
4. 在备服务器上从一个本地的WAL归档开始恢复， 如前所述等待使用 `recovery.conf` 所指定的 `restore_command` 。 （请参阅Section 24.3.4）。

恢复对WAL归档做只读处理，所以一旦在WAL的文件已被复制到备用系统， 就可以在同一时间复制到磁带，因为正通过备用数据库服务器读取。 因此，运行高可用性的备用服务器可以同时作为文件存储长远的灾难恢复目的做处理。

出于测试目的，它是可以在同一系统上运行的主备服务器。 没提供任何值得改进服务器的健壮性， 也不会描述为HA。

25.4.2. 基于记录的日志传送

使用这种替代方法也有可能实现基于记录的日志传送， 尽管这需要定制开发，一个完整的WAL文件传送之后变化只为热备查询可见。

一个外部程序可以调用 `pg_xlogfile_name_offset()` （参阅Section 9.26） 这个函数用来找出文件名和当前WAL结尾的准 确字节偏移。然后，可以直接访问WAL文件， 并从WAL的上次已知的结尾到当前结束数据复制数据到备用服务器。用这种方法， 数据丢失窗口是复制程序的轮询周期时间， 其可以非常小， 并没有迫使部分使用的段文件要归档的带宽浪费。 请注意备服务器上的 `restore_command` 脚本只能处理完整的WAL文件， 所以通常的增量备份数据到备服务器不可用。只有在主服务器死掉— 在允许它到来前，最后一部分WAL文件送到备服务器。在这个进程中的正确实现， 需要 `restore_command` 脚本与数据复制程序协作。

PostgreSQL9.0版本开始， 你可以使用流复制达到事半功倍的效果（请参阅Section 25.2.5）。

25.5. 热备

热备术语是用来形容连接到服务器，并运行只读查询的能力，而服务器在归档恢复或备模式。 对复制目的和非常精确的备份恢复到所需的状态，这是非常有用的。 长期的热备，也指从恢复到正常运行的服务器的能力， 而用户继续运行的查询和/或保持连接开放。

在热备用模式运行查询与正常的查询操作类似， 虽然有几个使用和管理的差异解释如下。

25.5.1. 用户概述

当备用服务器上hot_standby参数的设置为真时，将开始接受连接， 一旦恢复带来的系统到一致的状态。所有这些连接都严格只读的， 甚至可能没有可写的临时表。

数据从主服务器到备服务器上需要一些时间，所以会有一个主备数据库间的可测量的延迟。 因此，在主备服务器上几乎同时运行同样的查询返回不同的结果。 我们说在备服务器上的数据最终与主服务器上的一致。 一旦事务提交记录在备服务器是上回放， 由事务产生的变化对于在备服务器上的任何新快照来说是可见的。 快照可能是在每个查询或事务的开始，取决于当前事务的隔离级别。 请参阅Section 13.2获取更多的信息。

热备期间开始的事务可能会发出下面的命令：

- 查询访问- `SELECT` ， `COPY TO`
- 游标命令- `DECLARE` ， `FETCH` ， `CLOSE`
- 参数- `SHOW` ， `SET` ， `RESET`
- 事务管理命令
  - `BEGIN` , `END` , `ABORT` , `START TRANSACTION`
  - `SAVEPOINT` , `RELEASE` , `ROLLBACK TO SAVEPOINT`
  - `EXCEPTION` 阻塞其它内部的子事物。
- `LOCK TABLE` 仅当明确这些模式之一: `ACCESS SHARE` , `ROW SHARE` 或者 `ROW EXCLUSIVE` 。
- 规划和资源 - `PREPARE` , `EXECUTE` , `DEALLOCATE` , `DISCARD`
- 插件和扩展 - `LOAD`

在热备期间开始的事务，将从不会分配事务ID，并且不能写入到系统预写日志。 因此，以下操作将产生错误信息：

- 数据操纵语言(DML) - `INSERT` , `UPDATE` , `DELETE` , `COPY FROM` , `TRUNCATE` 。 请注意，不允许操作在恢复期间正执行触发器的结果。 此限制也适用于临时表，因为不分配一个事务ID，不能读取或写入表行， 在一个热备环境这种情况是不可能的。
- 数据定义语言(DDL) - `CREATE` , `DROP` , `ALTER` , `COMMENT` 。 甚至临时表也适用这个限制，因为执行这些操作将需要更新系统空间表。
- `SELECT ... FOR SHARE | UPDATE` 因为行锁，不能不采取更新底层数据文件。
- 在 `SELECT` 语句上的规则产生DML命令。
- `LOCK` 明确要求一个高于 `ROW EXCLUSIVE MODE` 的模式。
- `LOCK` 简短的缺省形式，因为它请求 `ACCESS EXCLUSIVE MODE` .
- 事务管理命令明确设置非只读状态：
  - `BEGIN READ WRITE` , `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE` , `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- 两阶段提交命令 - `PREPARE TRANSACTION` , `COMMIT PREPARED` , `ROLLBACK PREPARED` 因为即使只读事务需要在准备阶段写WAL。（两种阶段提交的第一个阶段）。
- 序列更新 - `nextval()` , `setval()`
- `LISTEN` , `UNLISTEN` , `NOTIFY`

在正常的操作，允许"只读"事务更新序列，使用 `LISTEN` , `UNLISTEN` 和 `NOTIFY` ， 所以热备会话下操作会比通常的只读会话限制稍微更严格。 在将来的版本中这些限制中的一些可能会放宽。

热备间， `transaction_read_only` 这个参数总为真，可能不会变。 但只要没有试图修改数据库，在热备的连接，将行动就像任何其它的数据库连接。 如果发生失效切换或倒换,数据库将切换到正常的处理模式。 当服务器改变模式， 会话将保持连接。一旦热备完成，有可能初始化读写事务（即使从热备间的会话）。

通过发出的 `SHOW transaction_read_only` 将告诉用户他们的会话是否只读的。 另外，一组函数允许用户访问关于备服务器的信息（请参阅Table 9-61） 这些允许你写程序获知数据库的当前状态。这些可以用来监视恢复进程， 或允许你写复杂的程序来恢复数据库到特定状态。

25.5.2. 处理查询冲突

主备服务器是许多方式松散连接的。在主服务器上的活动将在备服务器上生效。 作为一个结果，它们之间有潜在的负面交互或冲突.最容易理解的冲突是性能： 如果在主服务器上发生大数据量加载，然后将在备服务器上产生类似的WAL记录流， 所以备服务器查询可能竞争系统资源，像I/O。

在热备也可能发生额外的类型冲突。在该场景下，这些冲突是硬冲突。 可能需要取消查询，在某些情况下，为了解决它们，断开连接。 给用户提供几种解决这些冲突的方法。冲突情况包括：

- 在主服务器上采取访问排斥锁，包括明确的 `LOCK` 命令和多种DDL操作，在备服务器查询访问表冲突。
- 在主服务器上删除表空间与备服务器查询使用该空间的临时工作文件冲突。
- 在主服务器上删除一个数据库与在备服务器上连接到那个数据库的会话冲突。
- 一个从WAL清空记录的应用程序vacuum与在备服务器上事务，其快照仍然可以"看到"已删除的行。
- 一个从WAL清空记录的应用程序vacuum与在备服务器上查询访问该目标页，不管要删除的数据是否可见。

在主服务器上，这些情况简单等待结果，用户可能选择取消任何冲突的操作。 尽管，在备服务器上没有选择： 在主服务器上已经发生的WAL日志， 所以备服务器应用它一定不会失败。此外，允许WAL应用无限期等待可能是很不明智的。 因为备服务器的状态将变为增量远落后主服务器的。因此，提供一个机制， 强行取消备服务器上与将要应用WAL记录冲突的查询。

一个该问题情况的例子是管理员在主服务器上运行 `DROP TABLE` 一张表，而备服务器当前正查询这张表。 如果在备服务器上执行了 `DROP TABLE` ，明确的备服务器查询不能继续。 如果这个问题情况发生在主服务器。则 `DROP TABLE` 将等到其它查询完成。 但是当 `DROP TABLE` 运行在主服务器时，主服务器不会有关于备服务器查询的信息， 因此，将不等待任何备服务器查询。当备服务器查询在运行时，WAL改变的记录来到备服务器， 导致一个冲突。备服务器要么延迟应用WAL记录（任何事情也都要在它们之后）， 不然取消冲突的查询，由此可以应用 `DROP TABLE` 。

当一个冲突查询短的，通常想要允许它完成而延迟WAL应用程序一点点。 但是长时间的延迟WAL应用程序通常不是想要的。 所以取消机制有参数max_standby_archive_delay和max_standby_streaming_delay， 这定义在WAL应用程序中允许延迟最大值。一旦查询冲突比应用任何新收取的WAL数据设定有关延迟长， 则取消查询冲突。 有两个参数，因此有两个不同延迟，为从归档读取WAL数据 （即从一个基准备份初始化恢复或"赶上"已经远落后的备服务器） 和通过流复制读取WAL数据的指定延迟。

在备服务器存在高可用性的主服务器，最好设置延迟参数相对短， 因此不会由备服务器查询所导致延迟使远落后主服务器。 不过，如果备服务器意思为执行长时间的查询，那么一个高的或无期限的延迟值是可取的。 请记着如果延迟WAL记录应用程序，则长时间查询将导致 备

服务器上的其它会话不能看到最新的变化。

一旦超过了由 `max_standby_archive_delay` 或者 `max_standby_streaming_delay` 指定的延迟， 将取消查询冲突。 这通常结果是一个取消错误，虽然在回放 `DROP DATABASE` 整个数据库的情况下， 将终止冲突会话。此外，如果冲突由空闲事务保持，终止冲突会话。 （这个行为可能在将来版本改变）。

可能立即重试已取消的查询（在开始一个新事务之后，当然）。 自查询取消依赖于WAL记录重播的本质，如果再次执行，已经取消的查询可能很成功。

请记住这些参数与从备服务器接收WAL数据开始所经过的时间比较。 允许备服务器上任何查询的宽期限，从不超过该延迟参数， 并且如果备服务器存在落后主服务器，那么期限的可能相当小。 如等待之前查询执行完成的结果，或不能跟上有大量的更新负载的结果。

备用的查询和WAL重放之间的冲突最常见的原因是"过早清除"。 通常，PostgreSQL允许清理老行版本，当没有事务需要根据MVCC的规则 看到他们保证数据的正确性。 然而，这条规则只适用于在主库上事务执行情况。 所以在主库上清理删除主库上事务仍然可见的行版本，这是可能的。

有经验的用户应注意行版本的清理和行版本冷冻将与备用查询冲突。手动运行 `VACUUM FREEZE` 可能引起甚至没有更新或删除行的表上的冲突。

用户应该清楚那些表， 在主服务器上定期和大量更新表将会很快导致取消备服务器上长时间运行的查询。 在这类情况下，对 `max_standby_archive_delay` 或者 `max_standby_streaming_delay` 设置一个有限值，类似于设置 `statement_timeout` 。

如果发现不能接受某些取消备服务器查询，补救存在的可能性。 第一个选项是设置参数 `hot_standby_feedback` ， 阻止 `VACUUM` 删除最近的死行， 所以清理冲突不会发生。如果你这样做，你应该知道这将延迟主服务器清理死行，其可能不想要的表膨胀结果。 不过这种情况清理不逊于如果备服务器查询直接运行在主服务器上， 并且你仍然得到卸载执行在备服务器上的好处。 在这种情况下 `max_standby_archive_delay` 必须是保持大的， 因为延迟WAL文件可能已经包含了备服务器查询想要的记录项。

另一个选项是在主服务器上增加vacuum_defer_cleanup_age， 从而将不会像通常很快的清理掉死行。 这将允许在备服务器上取消它们前，更多时间给执行查询， 无需设置一个高的 `max_standby_streaming_delay` 。 虽然用这种方法保证窗口的执行时间是有困难的， 因为 `vacuum_defer_cleanup_age` 在主服务器执行的事务中是可测的。

查询数取消，原因可以看作在备用服务器上使用 `pg_stat_database_conflicts` 系统视图。 `pg_stat_database` 系统视图也包含摘要信息。

25.5.3. 管理员概述

如果在 `postgresql.conf` 中 启用 了 `hot_standby` ， 并且目前有个 `recovery.conf` 文件， 该服务器将运行在热备模式。不过可能花些时间为允许的热备连接， 因为该服务器不接受连接直到完成足够的恢复能提供一致的状态，其查询能运行。 在这个期间，将带有一个错误信息拒绝客户端尝试连接。为确认该服务器起来了， 要么循环尝试从应用程序连接，或者在服务器日志里查看这些错误信息：

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

在主服务器上每个检查点一致的信息记录一次。 在主服务器上没有将 `wal_level` 设置为 `hot_standby` 时，当读取正在写的WAL时， 启用热备是不可能的。存在这些条件的两者也可能延迟达到一致性状态：

- 一个写事务有多于64个子事务
- 很长时间活动的写事务

如果你正运行基于文件日志传送（"暖备"）， 你可能需要等到下一个WAL文件到来，其尽可能长如在主服务器设置 `archive_timeout` 。

有些参数的设置在备服务器将需要重新配置，如果在主服务器改变了它们。 对于这些参数，备服务器上的值要大于或等于主服务器上的。 如果这些参数没有设置足够高，那么备服务器将拒绝启动。 提供了更高的值，重启该服务器再开始恢复。这些参数是：

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`

管理员选择合适的设置为max_standby_archive_delay和 max_standby_streaming_delay是很重要的。 根据业务的优先级，最好的选择有所不同。例如：如果服务器是主要任务， 作为高可用性的服务器，那么你想低延迟设置，也许设置为0，尽管这也是很积极的设置。 如果备服务器的任务作为决策支持的额外服务器，那么可能接受设置最大延迟为几个小时， 或甚至-1意味着永远等待查询完成。

在主服务器上写的事务状态"提示位"没有记录WAL日志，所以在备服务器上将或许再次重写该提示。 因此，备服务器将仍然进行写磁盘即使所有用户是只读的，数据值自身没有发生改变。 用户将仍然写大量排序的临时文件和 重新生成缓存的信息文件， 所以在热备模式数据库没有部分是真只读的。还要注意写到远程数据库使用dblink模块， 外部数据的操作使用PL函数仍然是可能的，尽管事务是本地只读的。

在恢复模式里，不接受下面类型的管理命令：

- 数据定义语言(DDL) - e.g. `CREATE INDEX`
- 权限和所有权 - `GRANT` , `REVOKE` , `REASSIGN`
- 维护命令 - `ANALYZE` , `VACUUM` , `CLUSTER` , `REINDEX`

再次，请注意在主服务器的“只读”模式事务中，允许这里的某些命令。

作为一个总结，你不能创建额外的索引，统计也不能仅在备服务器， 如果需要这些管理命令，应该在主服务器执行，并且最终这些变化将传播到备服务器。

`pg_cancel_backend()` 和 `pg_terminate_backend()` 将在用户后台工作，但是不启动进程，其执行恢复。 `pg_stat_activity` 将不显示为一个启动进程项，也不显示做恢复事务的活动。 作为一个总结， `pg_prepared_xacts` 在恢复中总是空。 如果你愿解决有疑问准备的事务， 在主服务器上查看 `pg_prepared_xacts` 和发出命令来解决这里的事务。

`pg_locks` 将显示由后台持有的锁。 `pg_locks` 也显示由启动进程所管理的虚拟事务， 其拥有由恢复正回放的事务所持有的 `AccessExclusiveLocks` 。 请注意该启动进程不需要锁定数据库变化，并且非 `AccessExclusiveLocks` 锁， 不会显示在启动进程的 `pg_locks` 里。它们只是推测存在。

Nagios插件check_pgsql将工作，因为用它检测存在的简单信息。 check_postgres监控脚本也将工作， 尽管有些报告值能给不同或迷惑的结果。 例如：上次清理时间将不会保持， 因为在备服务器没有清理发生。 运行在主服务器的清理，将仍然发送它们的改变到备服务器。

在恢复期间WAL文件控制命令将不工作，比如 `pg_start_backup` , `pg_switch_xlog` 等。

动态加载模块工作，包括 `pg_stat_statements` 。

在恢复中咨询锁将工作正常，包括死锁保护。 请注意咨询锁从不写WAL日志，所以对于一个咨询锁在主服务器上或回放WAL在备服务器上冲突不可能的。 在主服务器上需要一个咨询锁，在备服务器上已经初始化了一个类似咨询锁也是不可能的。 咨询锁只是与需要它们的服务器相关。

基于触发器的复制系统像Slony, Londiste和Bucardo将不在备服务器运行， 尽管在主服务器运行的很好，但变化不会发送到备服务器应用。WAL回放不是基于触发器的， 所以你不能从备服务器中传送到任何系统，其需要额外的写或依赖使用触发器。

不能分配新OID，尽管某些UUID生成器可能仍然工作，只要不依靠它们写新状态到数据库。

当前，在只读事务中不允许创建临时表，所以在某些情况下存在的脚本将运行不正确。 这个限制可能在以后的版本中放宽。这是一个SQL标准的兼容性和技术问题。

如果表空间是空， `DROP TABLESPACE` 只能成功。 有些备服务器用户可积极的通过 `temp_tablespaces` 参数使用 该表空间。如果在表空间有临时文件，取消所有活动的查询来确保删除临时文件，所以可以删除表空间， 可以继续WAL回放。

在主服务器上运行`DROP DATABASE`或者`ALTER DATABASE ... SET TABLESPACE`将产生一个WAL项，其将导致已连接到在备服务器上的那个数据库的所有用户，强制断开连接。这个动作立即发生，不管`max_standby_streaming_delay`的设置。请注意`ALTER DATABASE ... RENAME`不会断开连接的用户，在多数情况下忽视，不过如果某些方式依赖数据库名，可能在某些情况下导致一个程序混乱。

在正常（非恢复）模式，如果你发出`DROP USER`或者`DROP ROLE`对于一个有登录权限的角色，当那个用户仍然已经连接，那么不会发生什么对于已连接的用户-他们保持连接。不过该用户不能再连接。这个行为在恢复中也适用，所以在主服务器上`DROP USER`不能断开备服务器上该用户连接。

在恢复中统计采集器是活动的。所有扫描，读取，块，索引使用等，将在备服务器中记录。回放活动将不复制在主服务器上的影响，因此回放插入将不增加插入pg_stat_user_tables列。恢复开始删除该统计文件，所以主备服务器的统计将不同，认为这是个特性，而不是一个臭虫。

在恢复中自动清理是不活跃的。在恢复结束将正常启动。

在恢复中后台记录器是活动的，将执行重启点（类似于主服务器上的检查点）和正常块清理活动。这可能包含存储在备服务器上的提示信息更新。在恢复中接受`CHECKPOINT`命令，尽管执行一个重启点而不是一个新检查点。

25.5.4. 热备参数参考

各种参数已经在Section 25.5.2和 Section 25.5.3上面提到。

在主服务器上，可以使用参数wal_level和 vacuum_defer_cleanup_age。如果在主服务器上设置，max_standby_archive_delay和 max_standby_streaming_delay没有影响。

在备服务器,可以使用参数hot_standby, max_standby_archive_delay和 max_standby_streaming_delay。只要服务器保留在备模式，vacuum_defer_cleanup_age没有影响，尽管变为相关的，如果备服务器成为主服务器。

25.5.5. Caveats

有几个热备限制。这些可能在将来的版本中解决：

- 在哈希索引的操作，不会记录在目前的WAL日志，索引回放将不更新这些索引。
- 在做快照之前充分认识运行的事务是必需的。事务使用大量的子事务（当前大于64）将延迟只读连接的开始直到运行最长写事务完成。如果这种情况发生，说明信息将发送到服务器的日志。

- 对于备服务器查询的有效开始点是产生在主服务器上的每个检查点。 如果备服务器关机，当主服务器在关机状态， 不可能重进热备直到启动主服务器， 所以在WAL日志里产生进一步的开始点。在最常见的情况下这种情况不是一个问题， 它可能发生。一般地，如果主服务器关机，不再可用，那可能由于一个严重的失败， 需要将备服务器转化为新主服务器运行。 并且有特意取下主服务器的情况， 协调确保备服务器成为平滑的主服务器，也是标准的处理。
- 在恢复结束，由准备的事务持有的`AccessExclusiveLocks`需要锁表正常数量的条目的两倍。 如果你计划运行大量并发的准备事务，正常地用`AccessExclusiveLocks`或你计划一个 大事务用多个`AccessExclusiveLocks`， 建议你选择一个大的`max_locks_per_transaction`值，可能为在主服务器上两倍这个参数值。如果你设置`max_prepared_transactions`为0， 根本不需要考虑这个。
- 串行化事务隔离级别在热备份中仍然不可用（参阅Section 13.2.3 and Section 13.4.1获取更多信息）。 在热备份模式中尝试设置事务为串行化隔离级别将产生一个错误。

Chapter 26. 恢复配置

Table of Contents

这一章介绍`recovery.conf`中可用设置。它们仅用于恢复的时间。 它们必须为后续希望的恢复进行重置，一旦开始恢复，那么不能进行修改。

`recovery.conf`中的设置采用的是`name = 'value'`的形式。 每一行声明一个参数。(`#`)用于注释。 使用2个引号(`''`)，在参数值中嵌入一个单引号。

安装目录的`share/`路径下有一个简单的例子，`share/recovery.conf.sample`。

26.1. 归档恢复设置

`restore_command` (`string`)

检索WAL文件中已归档段的SHELL命令。对归档恢复来说这个参数是必须的， 但对流复制来说是可选的。 字符串中的任何一个 `%f` 是用归档检索中的文件名替换， 并且 `%p` 是用服务器上的复制目的地的路径上复制目的地的路径名替换。 （路径名是相对当前工作路径的，如客户端的data路径） 任意一个 `%r` 是用包含最新可用重启点的文件名替换。这是最早的文件，必须保留以转储，从而实现一致性， 因此这个信息可以用于截断归档至实现从当前转储中重启的最低要求。 `%r` 典型的只用于热备配置(参阅Section 25.2)。 `%%` 可以嵌入一个实际的 `%` 字符。

对命令来说，只有当成功时返回一个零退出状态是很重要的。 命令将被要求归档命令中没有出现的文件名； 当为要求是，必须返回非零。如：

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

`archive_cleanup_command` (`string`)

这个选项参数声明一个在每次重启时执行的shell命令。 `archive_cleanup_command` 为清理备库不需要的归档WAL文件提供一个机制。 任何一个 `%r` 由包含最新可用重启点的文件名代替。这是最早的文件， 因此必须保留以允许转储能够重新启动，因此所有早于 `%r` 的文件可以安全的移除。 这个信息可以用于删除归档至能满足从当前转储重启的最低要求。 对典型单备配置中的 `archive_cleanup_command` 而言， 经常使用pg_archivecleanup模块，比如：

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

然而需要注意的是，如果多个备服务器从相同的归档路径转储， 需要确保在任何一个备服务器不在需要之前，不能删除WAL文件。 在热备配置中，会明显的用到 `archive_cleanup_command` (参阅Section 25.2)。 通过 `%%` ，在命令中嵌入一个实际的 `%` 字符。

如果命令返回一个非0的退出状态，那么将写一个警告日志信息。

`recovery_end_command` (`string`)

这个参数是可选的，用于声明一个只在恢复完成时执行的SHELL命令。 `recovery_end_command` 的目的是为复制或恢复之后进行的清理动作提供一个机制。 `%r` 由包含最新可用重启点的文件名代替， 如在archive_cleanup_command中的那样。

如果命令返回一个非0的退出状态，那么将写一个警告日志信息， 并且数据库将会继续启动。如果命令被一个信号终止，数据库不会继续启动。

26.2. 恢复目标设置

`recovery_target_name` (`string`)

此参数声明命名的还原点， 创建`pg_create_restore_point()`继续恢复。 可以指定`recovery_target_name`, recovery_target_time或者 recovery_target_xid最多之一。 默认是恢复到WAL日志的结尾。

`recovery_target_time` (`timestamp`)

这个参数设置一个时间戳，达到这个时间戳时会继续进行恢复。 在大多数的`recovery_target_time`, recovery_target_name或者 recovery_target_xid都可以声明这个参数。 缺省是恢复到WAL日志的结尾。精确的停止点也受到recovery_target_inclusive的影响。

`recovery_target_xid` (`string`)

这个参数声明一个事务ID，达到这个ID号继续进行恢复。 需要注意的是，当在事务开始时，顺序分配事务ID，事务会以不同数字顺序结束。 将被恢复的事务是那些在一个指定事务前（或包含该事务）提交事务。 在大多数`recovery_target_xid`, recovery_target_name或者 recovery_target_time都可以声明参数。 缺省是恢复到WAL日志尾。精确的停止时间也受 recovery_target_inclusive的影响。

`recovery_target_inclusive` (`boolean`)

声明是否在指定恢复目标(`true`)之后停止，或在这(`false`)之前停止。 应用于 recovery_target_time 和recovery_target_xid， 无论哪个，都是为这个恢复声明的。这表示事务是否具有明确的目标提交时间或ID， 会分别被包含在恢复中。缺省是`true`。

`recovery_target_timeline` (`string`)

声明在一个指定时间线进行恢复。缺省是当前正在进行基础备份的时间线。 设置它为`latest`恢复归档中发现的最新时间线，这在备用服务器中是有用的。 只需要在复杂的重新恢复的情况下声明这个参数，在这种情况下， 需要返回一个在PITR之后需要达到的状态。 参阅Section 24.3.5获取更多详细信息。

`pause_at_recovery_target` (`boolean`)

当达到恢复目标时，指定是否恢复应该暂停。默认是真。 如果恢复目标是恢复最理想的点，这是为了允许查询被执行反对 检查数据库。暂停状态可以使用`pg_xlog_replay_resume()` （参见 Table 9-62）恢复，然后 使恢复结束。如果恢复目标不是所需的停止点，那么关闭服务器，更改 恢复目标设置为以后的目标并重新启动继续恢复。

如果不启用hot_standby，或者没有设置恢复目标，那么这个设置 不起作用。

26.3. 备用服务器设置

`standby_mode` (`boolean`)

声明是否需要启动PostgreSQL服务器为一个standby。 如果这个参数为 `on` ，在达到归档WAL尾时，服务器不会停止恢复， 但会通过使用 `restore_command` 抓取新的WAL段（和/或通过连接到主服务器， 如 `primary_conninfo` 设置声明的那样）来尝试继续恢复。

`primary_conninfo` (`string`)

为连接到主服务器的备服务器声明一个连接字符串。 这个字符串的格式在Section 31.1.1中描述， 如果字符串中没有声明选项， 那么会检查相关的环境变量（参阅Section 31.14）。 如果环境变量也没有设置，那么使用缺省的。

连接字符串应该声明主库的主机名（或地址），以及端口号（如果与备库的缺省端口不同）。 同样的，声明一个用户名对应主库上具有合适权限的角色 （参阅Section 25.2.5.1）。 如果主库要求密码验证，那么还需要提供一个密码。 可以在 `primary_conninfo` 字符串中提供， 或者在备库上一个单独的 `~/.pgpass` 文件中（以 `replication` 作为数据库名）。 不用在 `primary_conninfo` 字符串中声明数据库名。

如果 `standby_mode` 设置为 `off` 时，则这个设置不起作用。

`trigger_file` (`string`)

指定一个触发器文件，用于在备库中结束恢复。 即使不设置此值，你还可以使用 `pg_ctl promote` 促进备份。 如果 `standby_mode` 为 `off` ，则这个设置不起作用。

Chapter 27. 监控数据库的活动

Table of Contents

一个数据库管理员常常想知道"现在系统正在干什么呢?"。 本章讨论如何回答这个问题。

有一些工具可以用来监控数据库活动以及分析性能。 本章大部分内容是用于描述PostgreSQL的统计收集器， 但我们也不能忽视普通的Unix监控程序， 比如 `ps` ， `top` ， `iostat` 和 `vmstat` 。 同样，一旦发现了某个性能恶劣的查询， 可能还要用PostgreSQL的EXPLAIN 命令进行进一步分析。 Section 14.1里讨论了 `EXPLAIN` 和其它用于理解独立查询行为的方法。

27.1. 标准Unix工具

PostgreSQL在大多数平台上修改 `ps` 输出的命令标题， 这样我们就很容易找出某个服务器进程。 一个简单的显示如下：

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: writer proces
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: wal writer pr
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum la
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats collect
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug 12
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl regressio
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl regressio
```

调用 `ps` 的方法因平台的不同而略有不同，显示出来的细节也有一些区别。 这个例子来自一个最近的Linux系统。这里显示出来的第一个进程是主服务器进程。 显示的命令参数和启动它的命令行参数相同。 下面是由主服务器进程自动调用的五个统计收集器后台进程， 如果你设置了系统不启动统计收集器，那么它们不会出现。 同样的可用禁用"autovacuum发射器"。 剩下的都是一个个处理客户连接的服务器进程，每个这样的进程都用下面的形式显示：

```
postgres: _user_ _database_ _host_ _activity_
```

在该客户端连接的生命期中，用户，数据库和(客户端)主机都保持不变， 但是活跃性指示符会变化。活跃性可以是 `空闲` (等待客户端的命令)、 `事务空闲` (在一个 `BEGIN` 块里等待用户)、或者一个命令类型名，比如 `SELECT` 。 同样，如果当前正在等待一个其它服务器进程持有的锁的时候， 会在信息后面附加 `waiting` 。在上面的例子中，我们可以推出： 进程15606正在等待15610完成其事务，这样才能释放一些锁。进程15610阻塞， 因为没有其他活动会话。在更复杂的情况下，有必要查看 `pg_locks` 系统视图来决定谁正在阻止他们。

如果关闭了update_process_title那么活跃性指示符将不会变化， 并且进程标题仅在新进程被启动的时候设置一次。 在某些平台上这样做可以节省每个命令的开销，但在其它平台上却没有这种差异。

> **Tip:** Solaris需要特别的处理。 你必需使用 `/usr/ucb/ps` 而不是 `/bin/ps` 。 你还必需使用两个 `w` 标志，而不是一个。 另外，你最初调用 `postgres` 时 用到的命令行在 `ps` 状态显示中必须比 每个服务器进程显示的短。如果没满足这三个条件， 那么 `ps` 为每个服务器进程输出的将是最初的 `postgres` 的命令行。

27.2. 统计收集器

PostgreSQL的统计收集器是一个支持收集和汇报服务器活跃性信息的子系统。 目前，这个收集器可以给出对表和索引的访问计数， 包括磁盘块的数量和独立行的项。它还跟踪每个表中的行的总数， 每个表的过去的清理和分析时间。它也可以计算用户定义的函数的调用， 以及每个人的总花费的时间。

PostgreSQL还可以判断当前其它服务器进程正在执行的命令是什么。 这是一个收集过程中的独立设施。

27.2.1. 统计收集器配置

因为统计收集给查询处理增加了一些开销，所以该系统可以配置为启用或禁用统计收集。 这是由配置参数控制的，通常在 `postgresql.conf` 里设置 (参阅Chapter 18获取有关设置配置参数的细节)。

参数track_activities启动监测任何服务器进程执行的当前命令。

参数track_counts控制关于表和索引是否被统计。

参数track_functions实现了对用户定义的函数用法的追踪。

参数track_io_timing启动监控块读写次数。

通常这些参数在 `postgresql.conf` 中设置，因此它们作用于所有服务器进程， 但是我们也可以在独立的会话里用SET命令把它们打开或者关闭。 为避免普通用户把它们的活跃性隐藏不给管理员看， 只有超级用户允许用 `SET` 命令修改这些参数。

统计收集器通过临时文件将采集到的信息传递给其他的PostgreSQL进程。 这些文件存放在stats_temp_directory参数命名的目录中。 缺省是 `pg_stat_tmp` 。 为了提高性能， `stats_temp_directory` 参数可以指向一个基于RAM的文件系统， 降低物理I/O需求。当服务器关闭时， 统计数据的永久复本存储在 `global` 子目录中， 所以统计数据可以在服务器重新启动时保留。

27.2.2. 查看收集到的统计信息

有一些预定义的视图可以用于显示统计收集的结果， 在Table 27-1里列出。 另外，我们可以使用底层的统计函数制作自定义的视图。 正如Section 27.2.3中讨论的。

在使用统计观察当前活跃性的时候，你必须意识到这些信息并不是实时更新的。 每个独立的服务器进程只是在准备进入空闲状态的时候才向收集器传送新的块和行访问计数； 因此正在处理的查询或者事务并不影响显示出来的总数。同样， 收集器本身也最多每 `PGSTAT_STAT_INTERVAL` 毫秒 (缺省500，除非在编译服务器的时候修改过) 发送一次新的报告。因此显示总是落后于实际活动。 但是由 `track_activities` 收集的当前查询信息总是实时更新的。

另外一个需要着重指出的是，在请求服务器进程显示任何这些统计信息的时候， 它首先抓取收集器进程发出的最新报告，然后就拿这些数据作为所有统计视图和函数的快照， 直到它当前的事务结束。因此统计信息在当前事务的持续期间内显示静态信息。类似的， 每个进程的当前查询信息在该查询首次出现在事务中的时候就被收集了， 并且在整个事务过程中都显示相同的信息。这是一个特性，而不是一个臭虫， 因为这样就允许你在统计上执行几个查询并且对结果进行相关性检查而又不用担心这些数字会悄悄的变化。 但是如果你想看每个查询的最新结果，那么就要记住在事务块外面处理这些查询。 另外，你可以调用 `pg_stat_clear_snapshot` ()， 这将丢弃目前事务的统计数据快照（如有）。 下次使用统计信息将导致获取一个新的快照。

在视图 `pg_stat_xact_all_tables` ， `pg_stat_xact_sys_tables` ， `pg_stat_xact_user_tables` 和 `pg_stat_xact_user_functions` 上事务也可以看到自己的统计（未传送到收集器）。 这些数字不能作为上面所说的；相反他们在整个事务中不断更新。

Table 27-1. 标准统计视图

视图名称	描述
`pg_stat_activity`	每个服务器进程一行， 显示进程当前活动相关的信息，比如状态和当前查询。 参阅pg_stat_activity获取更多详情。
`pg_stat_bgwriter`	只有一行， 显示关于后端写进程活动的统计信息。参阅pg_stat_bgwriter 获取更多详细信息。
`pg_stat_database`	每个数据库一行， 显示数据库广泛的统计。参阅pg_stat_database获取更多详情。
`pg_stat_all_tables`	当前数据库每个表一行， 显示关于访问特定表的统计。参阅pg_stat_all_tables 获取更多详细信息。
`pg_stat_sys_tables`	和 `pg_stat_all_tables` 一样， 除了只显示系统表之外。
`pg_stat_user_tables`	和 `pg_stat_all_tables` 一样， 除了只显示用户表。
`pg_stat_xact_all_tables`	类似 `pg_stat_all_tables` ， 但是到目前为止当前事务中计算采取的行动 （这不包含在 `pg_stat_all_tables` 中以及相关视图中。） 活的列数以及死行和清理以及分析操作不在此视图中出现。
`pg_stat_xact_sys_tables`	和 `pg_stat_xact_all_tables` 相同， 除了只显示系统表。
`pg_stat_xact_user_tables`	和 `pg_stat_xact_all_tables` 相同， 除了只显示用户表。

pg_stat_all_indexes	当前数据库中的每个索引的每一行，显示关于访问特定索引的统计。参见pg_stat_all_indexes获取更多详情。
pg_stat_sys_indexes	和 pg_stat_all_indexes 一样，但只显示系统表上的索引。
pg_stat_user_indexes	和 pg_stat_all_indexes 一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库每个表一行，显示特定表关于I/O的统计，参阅pg_statio_all_tables 获取更多细节。
pg_statio_sys_tables	和 pg_statio_all_tables 一样，但只显示系统表
pg_statio_user_tables	和 pg_statio_all_tables 一样，但只显示用户表。
pg_statio_all_indexes	当前数据库每个索引一行，显示特定索引关于I/O的统计。参阅pg_statio_all_indexes获取更多细节。
pg_statio_sys_indexes	和 pg_statio_all_indexes 一样的，但是只显示系统表上的索引。
pg_statio_user_indexes	和 pg_statio_all_indexes 一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库每个序列一行，显示特定序列关于I/O的统计。参阅 pg_statio_all_sequences获取更多细节。
pg_statio_sys_sequences	和 pg_statio_all_sequences 一样，但只显示系统序列。因为目前没有定义系统序列，所以这个视图总是空的。
pg_statio_user_sequences	和 pg_statio_all_sequences 一样，但只显示用户序列。
pg_stat_user_functions	每一个跟踪函数一行，显示关于执行这个函数的统计。参阅pg_stat_user_functions 获取更多详情。
pg_stat_xact_user_functions	类似于 pg_stat_user_functions ，但是在当前事务中只调用计数（这不包含在 pg_stat_user_functions 中）。
pg_stat_replication	每WAL发送进程一行，显示关于复制到发送端的链接备用服务器的统计信息。参阅pg_stat_replication获取更多细节。
pg_stat_database_conflicts	每个数据库一行，显示关于备用服务器恢复冲突取消查询的统计信息。参阅pg_stat_database_conflicts获取更多信息。

针对每个索引的统计有利于判断哪个索引得到使用以及它们的效果。

pg_statio_ 视图有利于决定缓冲区高速缓存的有效性。当实际的磁盘数读取比缓冲区的数目小得多的时候，然后缓存满足大多数读请求而没有调用内核调用。然而，这些统计数据不提供整个过程：由于PostgreSQL 处理磁盘I/O的方式，不在PostgreSQL缓冲区缓存中的数据可能仍然位于 内核I/O缓存中，因此可能仍然被取出而不需要物理读。对获得更多PostgreSQL的I/O行为的详细信息感兴趣的用户 建议使用与操作系统工具结合的PostgreSQL统计收集，允许洞察I/O的内核处理。

Table 27-2. `pg_stat_activity` 视图

列	类型	描述
`datid`	`oid`	连接后端的数据库OID
`datname`	`name`	连接后端的数据库名称
`pid`	`integer`	后端进程ID
`usesysid`	`oid`	登陆后端的用户OID
`usename`	`name`	登陆到该后端的用户名
`application_name`	`text`	连接到后端的应用名
`client_addr`	`inet`	连接到后端的客户端的IP地址。 如果此字段是null， 它表明通过服务器机器上UNIX套接字连接客户端或者这是内部进程如autovacuum
`client_hostname`	`text`	连接客户端的主机名，通过 `client_addr` 的反向DNS查找报告。 这个字段将只是非空的IP连接，并且仅仅当启动log_hostname的时候。
`client_port`	`integer`	客户端用于与后端通讯的TCP端口号，或者如果使用Unix套接字，则为 `-1` 。
`backend_start`	`timestamp with time zone`	该过程开始的时间，比如当客户端连接服务器时。
`xact_start`	`timestamp with time zone`	启动当前事务的时间，如果没有事务是活的，则为null。如果当前查询是首个事务，则这列等同于 `query_start` 列。
`query_start`	`timestamp with time zone`	开始当前活跃查询的时间， 或者如果 `state` 是非 `活跃的` ， 当开始最后查询时。
`state_change`	`timestamp with time zone`	上次 `状态` 改变的时间
`waiting`	`boolean`	如果后端当前正等待锁则为真
`state`	`text`	该后端当前总体状态。可能值是： `活跃的` :后端正在执行一个查询。 `空闲的` :后端正在等待一个新的客户端命令。 `空闲事务` ：后端在事务中，但是目前无法执行查询。 `空闲事务(被终止)` :这个情况类似于 `空闲事务` ，除了事务导致错误的一个语句之一。 `快速路径函数调用` :后端正在执行一个快速路径函数。 `禁用` ：如果后端禁用track_activities，则报

		告这个状态。
query	text	该后端的最新查询文本。如果 状态 是 活跃的，此字段显示当前正在执行的查询。在所有其他情况中，这表明执行过去的查询。

`pg_stat_activity` 每个服务器进程有一行，显示进程当前活动的相关信息。

> **Note:** `waiting` 和 `state` 列是独立的。 如果一个后端处于 `活跃` 状态，它可能会或可能不会 `waiting` 。 如果这种情况是 `活跃的` 并且 `waiting` 是真， 它意味着正在执行一个查询，但在该系统中的某个地方被锁阻塞。

Table 27-3. `pg_stat_bgwriter` 视图

列	类型	描述
`checkpoints_timed`	`bigint`	执行的定期检查点数
checkpoints_req	bigint	执行的需求检查点数
`checkpoint_write_time`	`double precision`	花费在检查点处理部分的时间总量，其中文件被写入到磁盘，以毫秒为单位。
checkpoint_sync_time	double precision	花费在检查点处理部分的时间总量，其中文件被同步到磁盘，以毫秒为单位。
`buffers_checkpoint`	`bigint`	检查点写缓冲区数量
buffers_clean	bigint	后端写进程写缓冲区数量
`maxwritten_clean`	`bigint`	后端写进程停止清理扫描时间数，因为它写了太多缓冲区
buffers_backend	bigint	通过后端直接写缓冲区数
`buffers_backend_fsync`	`bigint`	后端不得不执行自己的 `fsync` 调用的时间数 （通常后端写进程处理这些即使后端确实自己写）
buffers_alloc	bigint	分配的缓冲区数量
`stats_reset`	`timestamp with time zone`	这些统计被重置的时间

`pg_stat_bgwriter` 视图总是有独立行，包含集群的全局数据。

Table 27-4. `pg_stat_database` 视图

列	类型	描述
datid	oid	数据库的OID
datname	name	这个数据库的名字
numbackends	integer	当前连接到该数据库的后端数。 这是在返回一个反映目前状态值的视图中唯一的列；自 上次重置所有其他列返回累积值。
xact_commit	bigint	此数据库中已经提交的事务数
xact_rollback	bigint	此数据库中已经回滚的事务数
blks_read	bigint	在这个数据库中读取的磁盘块的数量
blks_hit	bigint	高速缓存中已经发现的磁盘块的次数， 这样读取是不必要的（这只包括PostgreSQL缓冲区高速缓存，没有操作系统的文件系统缓存。
tup_returned	bigint	通过数据库查询返回的行数
tup_fetched	bigint	通过数据库查询抓取的行数
tup_inserted	bigint	通过数据库查询插入的行数
tup_updated	bigint	通过数据库查询更新的行数
tup_deleted	bigint	通过数据库查询删除的行数
conflicts	bigint	由于数据库恢复冲突取消的查询数量。（只在备用服务器发生的冲突）；参阅pg_stat_database_conflicts获取更多信息。
temp_files	bigint	通过数据库查询创建的临时文件数量。计算所有临时文件， 不论为什么创建临时文件（比如排序或者哈希）， 而且不管log_temp_files设置。
temp_bytes	bigint	通过数据库查询写入临时文件的数据总量。计算所有临时文件， 不论为什么创建临时文件， 而且不管log_temp_files设置
deadlocks	bigint	在该数据库中检索的死锁数
blk_read_time	double precision	通过数据库后端读取数据文件块花费的时间，以毫秒计算。
blk_write_time	double precision	通过数据库后端写入数据文件块花费的时间，以毫秒计算。
stats_reset	timestamp with time zone	这些统计最后被重置的时间

`pg_stat_database` 视图将包含集群中每个数据库的每一行， 显示数据库统计。

Table 27-5. `pg_stat_all_tables` 视图

列	类型	描述
`relid`	`oid`	表的OID
`schemaname`	`name`	此表的模式名
`relname`	`name`	表名
`seq_scan`	`bigint`	此表发起的顺序扫描数
`seq_tup_read`	`bigint`	顺序扫描抓取的活跃行数
`idx_scan`	`bigint`	此表发起的索引扫描数
`idx_tup_fetch`	`bigint`	索引扫描抓取的活跃行数
`n_tup_ins`	`bigint`	插入行数
`n_tup_upd`	`bigint`	更新行数
`n_tup_del`	`bigint`	删除行数
`n_tup_hot_upd`	`bigint`	HOT更新行数（比如没有更新所需的单独索引）
`n_live_tup`	`bigint`	估计活跃行数
`n_dead_tup`	`bigint`	估计死行数
`last_vacuum`	`timestamp with time zone`	最后一次此表是手动清理的（不计算 `VACUUM FULL` ）
`last_autovacuum`	`timestamp with time zone`	上次被autovacuum守护进程清理的表
`last_analyze`	`timestamp with time zone`	上次手动分析这个表
`last_autoanalyze`	`timestamp with time zone`	上次被autovacuum守护进程分析的表
`vacuum_count`	`bigint`	这个表被手动清理的次数（不计算 `VACUUM FULL` ）
`autovacuum_count`	`bigint`	这个表被autovacuum清理的次数
`analyze_count`	`bigint`	这个表被手动分析的次数
`autoanalyze_count`	`bigint`	这个表被autovacuum守护进程分析的次数

`pg_stat_all_tables` 视图将包含 当前数据库中每个表的一行（包括TOAST表）， 显示访问特定表的统计信息。 `pg_stat_user_tables` 和 `pg_stat_sys_tables` 视图 包含相同的信息，但是过滤只分别显示用户和系统表。

Table 27-6. `pg_stat_all_indexes` 视图

列	类型	描述
`relid`	`oid`	这个索引的表的OID
`indexrelid`	`oid`	索引的OID
`schemaname`	`name`	索引中模式名
`relname`	`name`	索引的表名
`indexrelname`	`name`	索引名
`idx_scan`	`bigint`	索引上开始的索引扫描数
`idx_tup_read`	`bigint`	通过索引上扫描返回的索引项数
`idx_tup_fetch`	`bigint`	通过使用索引的简单索引扫描抓取的活表行数

`pg_stat_all_indexes` 视图将包含 当前数据库中的每个索引行，显示访问特定索引的统计。`pg_stat_user_indexes` 和 `pg_stat_sys_indexes` 视图包含相同的信息， 但是过滤只是分别显示用户和系统索引。

索引可以通过简单的索引扫描或"位图"索引扫描进行使用。位图扫描中 几个索引的输出可以通过AND或者OR规则进行组合， 因此当使用位图扫描的时候，很难将独立堆行抓取与特定索引进行组合， 因此，一个位图扫描增加 `pg_stat_all_indexes`.`idx_tup_read` 使用索引计数，并且增加 `pg_stat_all_tables`.`idx_tup_fetch` 表计数，但不影响 `pg_stat_all_indexes`.`idx_tup_fetch`。

> **Note:** `idx_tup_read` 和 `idx_tup_fetch` 计算不同甚至没有任何可使用的位图扫描。 因为 `idx_tup_read` 计算从索引检索的索引项而 `idx_tup_fetch` 计算从表抓取的活的行。如果任何死的或尚未提交的行使用索引进行抓取， 或通过唯一索引扫描避免任何堆抓取，则后者较小。

Table 27-7. `pg_statio_all_tables` 视图

列	类型	描述
`relid`	`oid`	表OID
`schemaname`	`name`	该表模式名
`relname`	`name`	表名
`heap_blks_read`	`bigint`	从该表中读取的磁盘块数
`heap_blks_hit`	`bigint`	此表缓存命中数
`idx_blks_read`	`bigint`	从表中所有索引读取的磁盘块数
`idx_blks_hit`	`bigint`	表中所有索引命中缓存数
`toast_blks_read`	`bigint`	此表的TOAST表读取的磁盘块数（如果存在）
`toast_blks_hit`	`bigint`	此表的TOAST表命中缓冲区数（如果存在）
`tidx_blks_read`	`bigint`	此表的TOAST表索引读取的磁盘块数（如果存在）
`tidx_blks_hit`	`bigint`	此表的TOAST表索引命中缓冲区数（如果存在）

`pg_statio_all_tables` 视图将包含 当前数据库中每个表的一行（包括TOAST表）， 显示出特定表I/O的统计。 `pg_statio_user_tables` 和 `pg_statio_sys_tables` 视图包含相同的信息， 但是过滤分别只显示用户和系统表。

Table 27-8. `pg_statio_all_indexes` 视图

列	类型	描述
`relid`	`oid`	索引的表的OID
`indexrelid`	`oid`	该索引的OID
`schemaname`	`name`	该索引的模式名
`relname`	`name`	该索引的表名
`indexrelname`	`name`	索引名称
`idx_blks_read`	`bigint`	从索引中读取的磁盘块数
`idx_blks_hit`	`bigint`	索引命中缓存数

`pg_statio_all_indexes` 视图将包含当前数据库中的每个索引行， 显示特定索引的I/O的统计。 `pg_statio_user_indexes` 和 `pg_statio_sys_indexes` 视图包含相同的信息， 但是过滤分别只显示用户和系统索引。

Table 27-9. `pg_statio_all_sequences` 视图

列	类型	描述
`relid`	`oid`	序列OID
`schemaname`	`name`	序列中模式名
`relname`	`name`	序列名
`blks_read`	`bigint`	从序列中读取的磁盘块数
`blks_hit`	`bigint`	序列中缓存命中数

`pg_statio_all_sequences` 视图包含当前数据库中每个序列的每一行，显示特定序列关于I/O的统计。

Table 27-10. `pg_stat_user_functions` 视图

列	类型	描述
`funcid`	`oid`	函数OID
`schemaname`	`name`	此函数中的模式名
`funcname`	`name`	函数名
`calls`	`bigint`	被调用的函数次数
`total_time`	`double precision`	在这个函数以及调用的其他函数的总时间，以毫秒为单位。
`self_time`	`double precision`	在这个函数本身上用的总时间，不包含调用其他函数的，以毫秒为单位

`pg_stat_user_functions` 视图包含每个跟踪函数的行，显示关于函数执行的统计。track_functions参数控制真正跟踪的函数。

Table 27-11. `pg_stat_replication` 视图

列	类型	描述
`pid`	`integer`	WAL发送进程的ID
`usesysid`	`oid`	登陆到这个WAL发送进程的用户OID
`usename`	`name`	登陆到WAL发送进程的用户名
`application_name`	`text`	连接到这个WAL发送端的应用名
`client_addr`	`inet`	客户端连接到这个WAL发送端的IP地址，如果这个字段为null，它表明通过 服务器上Unix套接字连接客户端。
`client_hostname`	`text`	连接客户端的主机名，通过 `client_addr` 的反向DNS查找报告。 并且当启用log_hostname时，这个字段对于IP连接是非空的。
`client_port`	`integer`	客户端正在使用与WAL发送端连接的TCP端口号，或者如果使用Unix套接字则为 `-1` 。
`backend_start`	`timestamp with time zone`	这个进程开始时的时间，比如当客户端连接到这个WAL发送端时。
`state`	`text`	当前WAL发送端状态
`sent_location`	`text`	在这次连接上发送的上次事务日志位置
`write_location`	`text`	通过备用服务器写入到磁盘的上次事务日志位置。
`flush_location`	`text`	通过备用服务器刷新到磁盘的上次事务日志位置。
`replay_location`	`text`	备用服务器上重播到数据库的上次事务日志位置。
`sync_priority`	`integer`	这个备用服务器被选为同步备用的优先级。
`sync_state`	`text`	该备用服务器的同步状态

`pg_stat_replication` 视图包含每个WAL发送进程的每一行， 显示发送端连接备用服务器有关复制的统计。列出只直接连接的备用； 没有可用的下游备用服务器的信息。

Table 27-12. `pg_stat_database_conflicts` 视图

列	类型	描述
datid	oid	数据库的OID
datname	name	数据库名称
confl_tablespace	bigint	由于删除的表空间，已经取消的数据库中的查询数量
confl_lock	bigint	由于锁超时，已经取消的数据库中的查询数
confl_snapshot	bigint	由于旧快照，已经取消的数据库查询数
confl_bufferpin	bigint	由于保留区而取消的数据库查询数
confl_deadlock	bigint	由于死锁已经被取消的数据库中的查询数

pg_stat_database_conflicts 视图将包含 每个数据库的一行，显示由于备用服务器恢复发生的冲突而取消查询的数据库范围统计信息。 这种视图将只包含备用服务器的信息，因为在主服务器上不会发生冲突。

27.2.3. 统计函数

查看统计的其他方式可以通过写查询设置，它使用相同的底层统计访问功能用于 上面显示的标准视图。函数名称的详细信息，请参考标准的视图定义。（例如， 在psql中你可以发出 \d+ pg_stat_activity 。） 每个数据库统计的访问函数以数据库OID作为参数识别报告给数据库。 每个表和索引函数看成表或索引的OID。每个函数统计采取一个函数OID。 请注意，只有当前数据库中的表，索引，和函数可以看出这些功能。

统计收集相关的附加函数列在Table 27-13中。

Table 27-13. 附加统计函数

函数	返回类型	描述
`pg_backend_pid()`	`integer`	服务器进程处理当前会
`pg_stat_get_activity`(integer`)	`setof record`	如果声明为 `NULL`， 则端信息记录，或者系统的记录。 返回的字段是 `pg_stat_activity` 视
`pg_stat_clear_snapshot()`	`void`	丢弃当前数据库快照
`pg_stat_reset()`	`void`	所有当前数据库统计计（需要超级用户权限）
`pg_stat_reset_shared (text)`	`void`	重置一些集群范围统计于参数（需要超级用户示在 `pg_stat_bgwriter` 调用 `pg_stat_reset_share` 归零。
`pg_stat_reset_single_table_counters (oid)`	`void`	为当前数据库中单一表计为零（需要超级用户
`pg_stat_reset_single_function_counters (oid)`	`void`	当前数据库中单一功能（需要超级用户权限）

`pg_stat_get_activity`， `pg_stat_activity` 视图的基本功能返回 包含每个后端进程的所有可用信息的记录集。有时获取这些信息的子集更方便。 在这种情况下，可以使用每个后端统计访问函数的旧设置。这些都显示在 Table 27-14中。这些访问函数使用后端ID号， 其范围从一到当前活动后端数。 函数 `pg_stat_get_backend_idset` 提供便利方式产生调用这些函数的每个活动后端的每一行。 比如，显示PID以及所有后端的当前查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Table 27-14. 每个后端统计函数

函数	返回类型	描述
`pg_stat_get_backend_idset()`	`setof integer`	设置当前活动的后端ID号（从1到活动后端数）
`pg_stat_get_backend_activity(integer)`	`text`	后端最近查询文本
`pg_stat_get_backend_activity_start(integer)`	`timestamp with time zone`	最近查询开始时间
`pg_stat_get_backend_client_addr(integer)`	`inet`	连接后端的客户端IP地址
`pg_stat_get_backend_client_port(integer)`	`integer`	客户端用于通讯的TCP端口号
`pg_stat_get_backend_dbid(integer)`	`oid`	连接后端的数据库OID
`pg_stat_get_backend_pid(integer)`	`integer`	后端进程ID
`pg_stat_get_backend_start(integer)`	`timestamp with time zone`	进程开始时间
`pg_stat_get_backend_userid(integer)`	`oid`	登陆到后端的用户OID
`pg_stat_get_backend_waiting(integer)`	`boolean`	如果后端正等待锁则为真
`pg_stat_get_backend_xact_start(integer)`	`timestamp with time zone`	当前事务的开始时间

27.3. 查看锁

监控数据库活动的另外一个有用的工具是 `pg_locks` 系统表。 这样就允许数据库管理员查看在锁管理器里面锁的信息。 比如，这个功能可以用于：

- 查看当前所有锁，所有在某一特定数据库里的关系上的锁， 所有在特定关系上的锁，或者某一PostgreSQL会话持有的所有锁。
- 判断当前数据库里带有最多未批准锁的关系(它很可能是数据库客户端的竞争源)。
- 判断锁竞争给数据库性能带来的影响， 以及锁竞争随着整个数据库流量的变化所产生的变化。

`pg_locks` 视图的细节在节Section 47.59里。 有关更多PostgreSQL的锁和管理并发性的信息， 请参考Chapter 13。

27.4. 动态跟踪

PostgreSQL允许对数据库服务器进行动态跟踪。 这样就允许在代码内特定的点上调用外部工具来跟踪执行过程。

许多跟踪点(也被称为"探头")已经插入在源代码中了， 这些探针的目的是被用于数据库开发者和管理员，默认情况下， 探头不编译成PostgreSQL； 用户必须运行配置脚本时明确启用它们。

目前，只有DTrace支持实用工具，在写这的时候， 它可在Solaris, Mac OS X, FreeBSD, NetBSD和Oracle Linux上使用。 SystemTap 项目为Linux还提供了一个DTrace的等效并且也是可用的。 通过改变 `src/include/utils/probes.h` 中的宏命令定义为支持其他的动态跟踪工具在理论上是可能的 。

27.4.1. 编译动态跟踪支持

跟踪点是默认禁止的，你必须明确告诉配置脚本以使得PostgreSQL中的探头可用。 使用 `--enable-dtrace` 选项来启用DTrace支持。 参见Section 15.4获取更多信息。

27.4.2. 内置跟踪点

Table 27-15显示的是在源代码中提供的标准跟踪点， Table 27-16显示探测中使用的类型。 更多探测可以被添加以提高PostgreSQL的观测性。

Table 27-15. 内置DTrace跟踪

名字	参数	描述
transaction-start	(LocalTransactionId)	开始新的事务触发探测器。arg0是事务ID。
transaction-commit	(LocalTransactionId)	当事务成功完成时触发探测器，arg0是事务ID。
transaction-abort	(LocalTransactionId)	当事务未成功完成时触发探测器，arg0是事务ID。
query-start	(const char *)	开始查询处理时触发探测器，arg0是查询字符串。
query-done	(const char *)	当完成查询处理时触发探测器，arg0是查询字符串。
query-parse-start	(const char *)	当开始查询解析时触发探测器，arg0是查询字符串。

query-parse-done	(const char *)	查询解析完成时触发探测器，arg0是查询字符串。
query-rewrite-start	(const char *)	启动查询重写时触发探测器。arg0是查询字符串。
query-rewrite-done	(const char *)	当查询重写完成时触发探测器，arg0是查询字符串。
query-plan-start	()	查询规划开始时触发探测器。
query-plan-done	()	查询规划完成时触发探测器。
query-execute-start	()	执行规划开始时将触发的探测器
query-execute-done	()	执行规划完成时将触发的探测器
statement-status	(const char *)	服务进程随时更新 `pg_stat_activity . status` 时触发的探测器。 arg0是一个新的状态字符串
checkpoint-start	(int)	检查点开始时触发的探测器。arg0可以逐位标记以区分不同的检查点类型， 如；shutdown，immediate， 或force。
checkpoint-done	(int, int, int, int, int)	检查点完成时触发的探测器（触发探测器列出检查点处理过程序列中的下一个探测器）。 arg0表示要写入的缓冲区的数目。arg1表示总的缓冲区的数目。 arg2，arg3和arg4包含了增加， 删除和循环回收的xlog文件的数目。
clog-checkpoint-start	(bool)	一个检查点的CLOG部分开始时触发的探测器。arg0对正常检查点是真， 对关闭检查点是假。
clog-checkpoint-done	(bool)	当一个检查点的CLOG部分完成时触发的探测器。 arg0的含义与CLOG-checkpoint-start一样。
subtrans-checkpoint-start	(bool)	当一个检查点的SUBTRANS部分开始时触发的探测器。 arg0对正常检查点是真， 对关闭检查点是假。
subtrans-checkpoint-done	(bool)	当一个检查点的SUBTRANS部分完成时触发的探测器。 arg0的含义与SUBTRANS-checkpoint-start一样。
multixact-checkpoint-start	(bool)	当一个检查点的MultiXact部分开始时触发探测器。 arg0对正常检查点表示真， 对关闭检查点表示假。
multixact-checkpoint-	(bool)	当一个检查点的MultiXact部分完成时触发的探测器， arg0的含义与multixact-checkpoint-start

done		一样。
buffer-checkpoint-start	(int)	开始一个检查点的缓冲区写部分时触发的探测器。 arg0持有逐位标识以区分不同的检查点类型，如shutdown，immediate或force。
buffer-sync-start	(int, int)	检查点期间，开始写脏缓冲区时触发的探测器（在识别出那个缓冲区必须写之后）。 arg0表示总缓冲区数，arg1表示当前脏的，需要写的缓冲区数。
buffer-sync-written	(int)	在检查点期间，每个缓冲区都被写了之后触发的探测器，arg0表示缓冲区的ID号。
buffer-sync-done	(int, int, int)	当所有脏缓冲被写之后触发的探测器。 arg0表示总缓冲区的数目。 arg1表示检查点进程实际写的缓冲区数。 arg2表示期望写的数目(arg1的buffer-sync-start)；任何的不同会导致另一个进程在检查点发生时刷新缓冲区。
buffer-checkpoint-sync-start	()	当完成将脏缓冲区写入到内核，并且还没有发出fsync请求之前触发的探测器。
buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探测器
twophase-checkpoint-start	()	当一个检查点的两相阶段状态部分开始时触发的探测器。
twophase-checkpoint-done	()	当一个检查点的两相阶段状态部分完成时触发的探测器。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当开始一次缓冲区读时触发的探测器。 arg0和arg1包含page块的锁和派生的子进程数（如果是一个关系扩展请求，arg1会是-1）。 arg2，arg3和arg4包含表空间，数据库和关系OID，以识别关系。 arg5是为局部缓冲创建临时关系时后端ID，或者共享缓冲区InvalidBackendId (-1)。 arg6对关系扩展请求表示真， 对正常读表示假。
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当完成一次缓冲区读时触发的探测器。 arg0和arg1包含page块的锁和派生的子进程数 （如果是一个关系扩展请求，arg1会表示新增锁的数目）。 arg2，arg3和arg4包含表空间，数据库和关系OID，以识别关系。 arg5是为局部缓冲创建临时关系时后端ID，或者共享缓冲区InvalidBackendId (-1)。 arg6对关系扩展请求表示真， 对正常读表示假。如果池中有缓冲区，则arg7表示真，反之表示假。
	(ForkNumber,	在发出共享缓冲区的任意写入请求时触发的探

buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	测器。 arg0和arg1包含分叉和页中块数。 arg2，arg3和arg4包含表空间，数据库和关系OID，以识别关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当完成一条写要求时触发的探测器。 需要注意的是， 它只影响将数据传递到内核参数的时间 ； 实际上，它不会写到磁盘上。这个参数与buffer-flush-start一致。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当服务器进程开始写脏缓冲区时触发的探测器。 如果经常发生，表示shared_buffers太小， 或需要调整bgwriter控制参数。 arg0和arg1包含分叉和页中的块数。 arg2，arg3和arg4包含表空间， 数据库和关系OID，以识别关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当完成脏缓冲区写时触发的探测器。参数与buffer-write-dirty-start一样。
wal-buffer-write-dirty-start	()	当服务器进程开始写脏WAL缓冲时触发的探测器（此时WAL缓冲区已满）。 如果经常发生，应该是wal_buffers设置的太小了。
wal-buffer-write-dirty-done	()	当完成一次脏WAL写时触发的探测器。
xlog-insert	(unsigned char, unsigned char)	当插入一条WAL记录时触发的探测器。 arg0表示记录的rm id。 arg1包含信息标志。
xlog-switch	()	当要求进行WAL切换时触发的探测器。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	开始从一个关系中读取锁时触发的探测器。 arg0和arg1包含page块的锁和派生的子进程数。 arg2，arg3和arg4包含表空间，数据库和关系OID， 以识别关系。arg5是为局部缓冲创建临时关系时的后端ID，或者共享缓冲InvalidBackendId (-1)
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个锁读取完成时触发的探测器。 arg0和arg1包含page块的锁和派生的子进程数。 arg2，arg3和arg4包含表空间， 数据库和关系OID，以识别关系。 arg5是为局部缓冲创建临时关系时的后端ID，或者共享缓冲InvalidBackendId (-1)， 而arg6表示实际读取的字节数，而arg7是要求数（如果不一样会报错）
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当向一个关系中写入锁时触发的探测器。 arg0和arg1包含page块的锁和派生的子进程数。 arg2，arg3和arg4包含表空间，数据库和关系OID，以识别关系。 arg5是为局部缓冲创建临时关系时的后端ID，或者共享缓冲InvalidBackendId (-1)。

smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个锁写进程完成时触发的探测器。 arg0和arg1表示page块的锁和派生的子进程数。 arg2，arg3和arg4包含表空间，数据库和关系OID，以识别关系。 arg5表示为局部缓冲创建临时关系时的后端ID，或者共享缓冲InvalidBackendId (-1)， 而arg6表示实际读取的字节数，而arg7是要求数（如果不一样会报错）。
sort-start	(int, bool, int, int, bool)	排序操作开始时触发的探测器。 arg0表示堆，索引或者基准点。 arg1对强制唯一值表示真。 arg2表示键列的数目。 arg3表示允许使用的内存数目（以千字节为单位）。 如果要求随机访问排序结果，那么arg4表示真。
sort-done	(bool, long)	排序操作结束时触发的探测器。 arg0对外部排序表示真，内部排序表示假。 arg1表示用于一个外部排序的磁盘锁的数目， 或用于一个内部排序的，以千字节为单位的内存数目。
lwlock-acquire	(LWLockId, LWLockMode)	当成功获得一个LWLock时触发的探测器。 arg0是LWLock的ID号，arg1表明请求的锁模式，要么独占要么共享
lwlock-release	(LWLockId)	LWLock释放时触发的探测器 (但是请注意任何发布的等待者还未觉醒)。 arg0表示LWLock的ID号
lwlock-wait-start	(LWLockId, LWLockMode)	当不能立即获得LWLock锁，同时服务进程进入等待时触发的探测器。 arg0是LWLock的ID号，arg1表明请求的锁模式，要么独占要么共享。
lwlock-wait-done	(LWLockId, LWLockMode)	当从一个LWLock锁中释放服务进程时触发的探测器（实际上没有进行锁）。 arg0是LWLock的ID号，arg1表明请求的锁模式，要么独占要么共享。
lwlock-condacquire	(LWLockId, LWLockMode)	当成功获得一个LWLock时触发的探测器（已声明调用无需等待）。 arg0是LWLock的ID号，arg1表明请求的锁的模式，要么独占要么共享。
lwlock-condacquire-fail	(LWLockId, LWLockMode)	当没有成功获得一个LWLock时触发的探测器（已声明调用无需等待）。 arg0是LWLock的ID号，arg1表明请求的锁的模式，要么独占要么共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr锁）的请求开始等待（因为无法获得锁）时触发的探测器。 arg0到arg3是辨别被锁定对象的标签字段。arg4指出被锁对象的类型。 arg5表示请求的锁类型。
	(unsigned int, unsigned int,	

lock-wait-done	unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr锁）的请求结束等待时触发的探测器， 参数与lock-wait-start一样。
deadlock-found	()	当死锁探测器发现死锁时触发的探测器

Table 27-16. 定义用于探测器参数的类型

类型	定义
LocalTransactionId	unsigned int
LWLockId	int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

27.4.3. 使用跟踪点

下面的例子显示了一个分析事务次数的DTrace脚本， 可以用来代替性能测试之前和之后的 `pg_stat_database` 快照。

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

例如示范D脚本执行时，如输出：

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

> **Note:** SystemTap为跟踪脚本使用一个不同的标记而不是Dtrace， 即使底层的跟踪点是兼容的。有一点需要注意， 在这样写的时候，SystemTap脚本必须使用双下划线代替连字符来指向探测器名。 希望在未来SystemTap的版本中修复。

你应该记住DTrace脚本需要仔细的编写和充分的调试， 否则收集到的跟踪信息可能毫无意义。 大多数情况下问题是手段是错误的而不是底层系统。 在讨论使用动态跟踪发现的信息时， 应确保包含允许检查和讨论使用的脚本。

更多的示例脚本可以在PgFoundry dtrace project 中找到。

27.4.4. 定义新的跟踪点

开发者可以在代码中任意位置定义新的跟踪点， 当然这要重新编译之后才能生效。下面是用于新探测器插入步骤：

1. 通过探头决定探头名字和可利用数据。
2. 新增探头定义为 `src/backend/utils/probes.d`
3. 包括 `pg_trace.h` ， 如果已经不在模块中包含探测点， 并且在所需源代码中期望的位置插入 `TRACE_POSTGRESQL` 探测宏。
4. 重新编译和验证新探头是可用的。

例子: 下面是一个例子，你将如何添加一个探头通过事务ID追踪所有新的事务。

1. 决定探测器将被命名为 `transaction-start` 并且需要LocalTransactionId类型参数。
2. 新增探头定义为 `src/backend/utils/probes.d` ：

   ```
   probe transaction__start(LocalTransactionId);
   ```

   注意探测器名字中的双下划线的使用。在使用探测器的DTrace脚本中， 需要用一个连字符来替换双下划线， 因此， 对用户而言， `transaction-start` 是文档名。
3. 在编译时， `transaction__start` 被转换成一个宏调用 `TRACE_POSTGRESQL_TRANSACTION_START` （注意这里是单下划线）， 可以从 `pg_trace.h` 中获得。将宏调用放在源代码中的合适位置。 在这种情况下， 类似于下面：

```
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
```

4. 在重新编译和运行新的二进制文件之后， 通过运行下面的DTrace命令来检查新增的探测器是否可用。 应该得到类似下面的结果：

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           FUNCTION NAME
18705 postgresql49878     postgres     StartTransactionCommand transaction-start
18755 postgresql49877     postgres     StartTransactionCommand transaction-start
18805 postgresql49876     postgres     StartTransactionCommand transaction-start
18855 postgresql49875     postgres     StartTransactionCommand transaction-start
18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

向C代码中添加跟踪宏时，有一些注意事项，见下文：

- 需要注意的是，为探测器参数声明的数据类型要匹配宏中可用的数据类型， 否则会发生编译错误。
- 在大多数平台上，如果编译PostgreSQL 时带有 `--enable-dtrace` 选项， 无论何时通过宏来控制时，都会估算该跟踪宏的参数， 即使没有进行跟踪。通常不需要担心是否你只是报告一些局部变量的值。 但要注意将重要的函数调用放置在参数中。如果需要这么做，考虑通过检查是否真的开启跟踪来保护宏：

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

  每个跟踪宏都有一个相应的 `ENABLED` 宏。

Chapter 28. 监控磁盘使用情况

Table of Contents

本章讨论如何观察PostgreSQL数据库系统的磁盘使用情况。

28.1. 判断磁盘的使用量

每个表都有一个主堆(primary heap)磁盘文件，大多数数据都存储在这里。如果一个表存在值可能会很长的字段， 则另外还有一个用于存储因为数值太长而不适合存储在主表中的数据的TOAST文件 (参阅Section 58.2)。如果存在这个扩展表，那么将会同时存在一个TOAST索引。 当然，同时还可能有索引和基表关联。每个表和索引都存放在单独的磁盘文件里(超过1GB 可能会被分割成多个)。 这些文件的命名原则在Section 58.1里描述。

可以使用三种方法监视磁盘空间：使用Table 9-64中列出的 SQL 函数、 使用oid2name模块、或使用手动检查系统表。SQL 函数是最简单的方法并且一般推荐使用它。 本节其余部分显示了如何通过检查系统表来监视磁盘空间。

在最近刚刚清理(或者分析过)的数据库上使用psql的话， 可以使用查询来查看任意表的磁盘使用：

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

每个页通常都是 8K 字节。注意， `relpages` 只被 `VACUUM` ， `ANALYZE` 和几个 DDL 命令(例如 `CREATE INDEX`)更新。如果你想直接检查表的磁盘文件， 那么可以使用文件路径名。

要显示TOAST表使用的空间，我们可以使用一个类似下面这样的查询：

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT reltoastidxid
             FROM pg_class
             WHERE oid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

也可以很容易地显示索引的尺寸：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

很容易用下面的信息找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

28.2. 磁盘满导致的失效

一个数据库管理员最重要的磁盘监控任务就是确保磁盘不会写满。 磁盘写满可能不会导致数据的丢失，但它肯定会导致系统进一步使用的问题。 如果 WAL 文件也在同一个磁盘上(缺省配置就是这样)，则会发生数据库服务器恐慌， 并且停止运行。

如果你不能通过删除其它东西来释放磁盘空间， 那么你可以通过使用表空间把一些数据库文件移动到其它文件系统上去。 参阅Section 21.6获取更多信息。

> **Tip:** 有些文件系统在快要写满的时候性能会急剧恶化，因此不要等到磁盘完全写满时才采取行动。

如果你的系统支持针对每个用户的磁盘限额，那么数据库自然也将受制于此， 超过限额的影响和完全用光磁盘空间是完全一样的。

Chapter 29. 可靠性和预写式日志

Table of Contents

本章解释预写式日志如何用于获得有效性和可靠性操作。

29.1. 可靠性

可靠性是任何严肃的数据库系统的重要属性，PostgreSQL尽一切可能来保证可靠的操作。 可靠性操作的一个方面是所有已提交的数据都应该存储在一个非易失的区域，这样就不会因为电力失效、 操作系统崩溃、硬件失效(除了非易失区域自身失效之外)等原因导致数据丢失。 向计算机的永久存储 (磁盘驱动器或者等效的东西)成功写入数据通常可以满足这个要求。实际上，即使计算机完全失效， 只要磁盘驱动器生存下来，那么它们就可以移动到另外一台类似硬件的计算机上， 而所有已经提交的事务将保持原状。

周期性地强制数据写入磁盘盘片看上去像一件简单的操作，但实际上不是。 因为磁盘驱动器比内存和 CPU 要慢许多，在计算机的主存和磁盘盘片之间存在多层缓冲。 首先，有操作系统的缓冲区内存，它缓冲常用的磁盘块并且组合对磁盘写入的请求。幸运的是， 所有操作系统都给予应用一个强制从缓冲区写入磁盘的方法，PostgreSQL 使用了该特性(参阅wal_sync_method参数)。

然后，在磁盘驱动器的控制器上可能还有一个缓冲；特别是在RAID控制卡上更为常见。 这些缓冲区中，有些是透过式写入，意思是写入动作在到达的同时写入到磁盘上。 其它则是回写式写入，意思是数据将在稍后写入驱动器。这样的缓冲区是可靠性的危害， 因为磁盘控制器上的内存是易失的，在发生电力失效的情况下会丢失其中的内容。 好一些的控制器卡备有电池备份单元(BBUs)， 可以在系统电力失效的情况下提供电力。在电力恢复之后，这些数据将会被写入磁盘驱动器。

最后，大多数磁盘驱动器自身也有缓冲区。有些是透过式的，有些是回写式的。 和磁盘控制器一样，回写式的磁盘缓冲区也存在数据丢失的问题。 消费级别的 IDE 和 SATA 驱动器特别容易包含回写式缓冲，在掉电的情况下很容易丢失数据。 很多固态磁盘(SSD)也有易失的回写式缓冲。

这些缓存通常可以禁用；然而，这样做的方法会因操作系统和驱动类型而不同。

- 在Linux上，IDE和SATA驱动器可以使用 `hdparm -I` 查询； 如果在 `Write cache` 后面有一个 `*` 则写缓存是开启的。 `hdparm -W 0` 可以用来关闭写缓存。SCSI驱动器可以使用sdparm 查询。使用 `sdparm --get=WCE` 来检查写缓存是否打开， 使用 `sdparm --clear=WCE` 来关闭写缓存。
- 在FreeBSD上，IDE驱动器可以使用 `atacontrol` 查询， 写缓存在 `/boot/loader.conf` 里用 `hw.ata.wc=0` 来关闭； SCSI驱动器可以使用 `camcontrol identify` 查询，写缓存也同时查询， 并且当可用时使用 `sdparm` 改变状态。
- 在Solaris上，磁盘写缓存通过 `format -e` 来控制。 （Solaris ZFS文件系统对于打开磁盘写缓存是安全的，因为它发行自己的磁盘缓存刷新命令。）

- 在Windows上，如果 `wal_sync_method` 为 `open_datasync` （默认值）， 写缓存可以通过取消选中 `My Computer\Open\``_disk drive_` \Properties\Hardware\Properties\Policies\Enable write caching on the disk 来禁用。或者，设置 `wal_sync_method` 为 `fsync` 或 `fsync_writethrough` 来阻止写缓存。

- 在Mac OS X上，写缓存可以通过设置 `wal_sync_method` 为 `fsync_writethrough` 来阻止。

最近的SATA驱动器（跟随ATAPI-6或以后的）提供一个驱动器缓存刷新命令(`FLUSH CACHE EXT`)， 而SCSI驱动器长期以来支持一个类似的命令 `SYNCHRONIZE CACHE` 。这些命令不能直接访问PostgreSQL， 但是某些文件系统（例如，ZFS，ext4）可以使用它们来刷新数据到启用了回写的驱动器上的盘片上。 不幸的是，这样的文件系统只有在组合电池备份单元(BBU)磁盘驱动器时行为有效。在这样的设置中， 同步命令强制所有数据从驱动器缓存到磁盘，消除BBU的好处。你可以运行pg_test_fsync 程序来查看你是否受影响。如果你受到影响，BBU的性能优势可以通过在文件系统中关掉写屏障或重新配置磁盘控制器重新获得，如果这是一个选项。如果关闭了写屏障，确保电池仍然运行；失效的电池可能导致数据丢失。 希望文件系统和磁盘控制器设计将最终解决这个次优的行为。

在操作系统向存储硬件发出一个写请求的时候，它没有什么好办法来保证数据真正到达非易失的存储区域。 实际上，确保所有存储部件都保证数据和文件系统元数据的完整性是管理员的责任。 应该避免使用没有电池供电的回写缓冲磁盘控制器。在驱动级别， 如果驱动器不能保证在关闭(掉电)之前写入数据，那么应该关闭回写缓冲。 如果你使用SSD，要知道这些默认不尊重缓存刷新命令。你可以使用 `diskchecker.pl` 测试可靠的I/O子系统的行为。

另外一个数据丢失的风险来自磁盘盘片写操作自身。磁盘盘片会被分割为段，通常每段 512 字节。 每次物理读写都对整个段进行操作。当一个写操作到达磁盘的时候，它可能是512字节的某些整数倍 （PostgreSQL通常一次写入8192字节，或者16段），而写入操作可能因为电力失效而随时失败， 意味着某些 512 字节的段写入了，而另一些则没有。为了避免这个问题，PostgreSQL 在修改磁盘上的实际页面之前周期性地把整个页面的影像写入永久WAL存储。这样， 在崩溃恢复的时候，PostgreSQL就可以从WAL中恢复部分写入的页面。 如果你有文件系统(比如ZFS)自身能够避免部分页面写入，你可以通过关闭full_page_writes 参数来关闭页面影像功能。电池备份单元(BBU)磁盘控制器不能阻止部分页面写入， 除非他们保证数据以完整页面（8kB）写入BBU。

PostgreSQL也能阻止某些因为硬件错误或介质失效可能发生在存储驱动器上的数据损坏， 比如读/写垃圾数据。

- 在WAL文件中的每个单独的记录受CRC-32 (32-bit)校验保护，这样允许我们判断记录内容是否正确。 当我们在崩溃恢复时写入每个WAL记录和检查、归档记录和复制时设置CRC值。

- 数据页不是当前校验和的，尽管在WAL记录中的整个页面图像记录将受到保护。 为未来使用数据页校验和特性，数据页有一个16位字段可用。

- 内部数据结构例如 `pg_clog`，`pg_subtrans`，`pg_multixact`，`pg_serial`，`pg_notify`，`pg_stat`，`pg_snapshots` 不是直接校验和的， 也不是通过全页面写入受到保护的页面。然而，这样的数据结构具有持久性， WAL记录书面允许最近的改变在崩溃恢复时精确重建并且这些WAL记录正如上述讨论的那样受到保护。
- 在 `pg_twophase` 中的个人状态文件受到CRC-32保护。
- 临时数据文件用于较大的SQL查询，具体化和中间结果不是当前校验和的， 也不会WAL记录被写入修改这些文件。

PostgreSQL不预防矫正内存错误，假设您将使用具有工业标准纠错编码(ECC) 的内存操作或更好的保护。

29.2. 预写式日志(WAL)

预写式日志(WAL)是一种确保数据完整性的标准方法。 有关它的详细描述可以在大多数(如果不是全部的话)有关事务处理的书中找到。简而言之， WAL的中心思想是对数据文件的修改(它们是表和索引的载体) 必须是只能发生在这些修改已经记录到日志之后，也就是说， 在描述这些变化的日志记录刷新到永久存储器之后。如果我们遵循这个过程， 那么就不需要在每次事务提交的时候都把数据页刷新到磁盘， 因为在出现崩溃的情况下可以用日志来恢复数据库： 任何尚未附加到数据页的记录都将先从日志记录中重做(这叫向前滚动恢复，也叫REDO)。

> **Tip:** 因为WAL在崩溃之后恢复数据文件内容， 所以日志文件系统对于数据文件或WAL文件的可靠存储是完全没有必要的。实际上， 日志记录开销会降低性能，尤其是日志记录导致文件系统*data* 刷新到磁盘。幸运地，在记录日期期间的数据刷新可以经常用一个文件系统挂载选项禁用， 例如，在Linux ext3文件系统上的 `data=writeback` 。在崩溃后日志文件系统确实提高了启动速度。

使用WAL显著地减少了磁盘写的次数，因为只有日志文件需要刷新到磁盘以保证事务提交了， 而不是事务修改的所有数据文件。日志文件是顺序写的，所以同步日志的开销要远比同步数据页的开销小。 对于许多小事务修改数据存储的许多不同位置更是如此。另外，当服务器正在处理许多小的并发事务时， 日志文件的一个 `fsync` 足以提交许多事务。

WAL还提供了数据库在线备份和时间点恢复的可能，就像Section 24.3 里描述的那样。通过归档的 WAL 文件，可以将数据库恢复到 WAL 文件包含的任意时刻： 只需要简单地安装以前的数据库物理备份，然后重放 WAL 到希望的时间点。另外， 物理备份还不必是数据库状态的一个即时快照(如果其制作花了较长时间的话)， 因为 WAL 日志的重放将修复任何内部的不一致。

29.3. 异步提交

异步提交是一个允许事务更快完成的选项，如果数据库崩溃时，最新的事务可能会丢失。 在大多数应用中这个是可以接受的交易。

如在前一节中所述，事务提交通常是同步的：服务器等待事务的WAL 记录刷新到永久存储区，然后返回一个成功提示到客户端。客户端因此保证报告提交的事务将被保存， 即使服务器立即崩溃。然而，对于简短的事务，延迟是总事务时间的主要部分。选择异步提交模式意味着， 在它产生的WAL记录实际转移到磁盘之前，一旦事务逻辑上完成之后服务器就返回成功。 这样在小事务的吞吐量上可以提供一个显著的推动作用。

异步提交引进了数据丢失的风险。在向客户端报告事务完成和事务实际完成的时间点之间有一个很小的时间窗口 （也就是，保证如果服务器崩溃时不会丢失）。因此如果客户端采取外部动作（前提是假设事务会被记下）， 此时不应使用异步提交。例如，一个银行肯定不会为一个ATM分配现金的事务使用异步提交。但在多数情况下， 如事件日志记录，不需要这种强力保证。

使用异步提交的风险在于数据丢失，而不是数据损坏。如果数据库崩溃，那么它会根据刷入的最新的 WAL记录来进行恢复。因此数据库会被转储到一个一致的状态， 但任何没有写入到磁盘的事务不能反映在这个状态。因此最后的结果是丢失最新的几个事务。 因为事务是以提交的顺序进行重放，不会引入非一致状态；例如，如果事务B所做的更改依赖于前一个事务A， 那么丢失A的效果，而B的效果被保留是不可能的。

用户可以为每个事务选择提交模式，因此可以同时使用同步和异步提交事务。 这样就允许在性能和事务持久性之间做一个灵活的权衡。提交模式是由用户可设定的参数synchronous_commit控制的，可以用任意配置参数可以设置的方式改变。 用于任意一个事务的模式依赖于事务提交开始时 `synchronous_commit` 的值。

某些实用命令，如 `DROP TABLE` ，会强制使用同步提交，无论 `synchronous_commit` 参数是怎么设置的。这是为了保证服务器文件系统和数据库逻辑状态之间的一致性。支持两阶段提交的命令， 如 `PREPARE TRANSACTION` ，也是使用同步提交。

如果在一个异步提交和写事务的WAL记录中间时发生数据库崩溃， 在事务期间的修改将会丢失。风险的持续时间会被限制，因为"WAL writer" 后台进程会每隔wal_writer_delay毫秒就向磁盘刷入未写入的WAL记录。 风险的实际最大持续时间是三倍的 `wal_writer_delay` ， 因为WAL写进程被设计为支持在繁忙时一次写入所有块。

> **Caution**
>
> IMMEDIATE模式的关闭等同于服务器崩溃，因此会造成哪些未刷入的异步提交的数据丢失。

异步提交不同于设置fsync = off。 `fsync` 是一个用于更改所有事物行为的服务器端设置。它会禁用所有PostgreSQL 中尝试向数据库不同部分同步写入的逻辑，因此，一次系统崩溃（硬件或操作系统崩溃， 而不是PostgreSQL本身出问题）会导致数据库状态不可预知的崩溃。在多数情况下，通过关闭 `fsync` ，异步提交会提高性能，但是不会有数据崩溃的风险。

看起来，commit_delay与异步提交很相似，但实际上它是一种同步提交方法 （事实上，异步提交时会忽略 `commit_delay` ）。在一次异步提交尝试向磁盘刷入 WAL之前， `commit_delay` 会造成延迟， 希望在一个这样的事务中执行一个单独的刷入可以用于同一时间的其他事务提交。 这个设置可以看做一种在可以加入一个关于参与单次刷入的组中的事务中提高时间窗口的方式， 来分摊多个事务刷新的开销。

29.4. WAL 配置

有几个与WAL相关的参数会影响数据库性能。本节讨论它们的使用。 参阅Chapter 18获取有关服务器配置参数的一般信息。

检查点是事务序列中的点， 在该点之前的所有信息都确保已经写到数据文件中去了。在检查点时， 所有脏数据页都刷新到磁盘并且向日志文件中写入一条特殊的检查点记录。 （变更记录已经预先刷新到WAL文件了。）在发生崩溃的时候， 崩溃恢复过程查找最后的检查点记录，判断应该从日志中的哪个点(称为 redo 记录)开始 REDO 操作， 在该记录之前对数据文件的任何修改都保证已经写在磁盘上了。因此，在检查点之后， 任何在包含 redo 记录点之前的日志段都不再需要，因此可以循环使用或者删除。当然， 在进行WAL归档的时候， 这些日志段在循环利用或者删除之前必须先归档。

刷新所有脏数据页面到磁盘的检查点需求会导致显著的I/O负载，因此，检查点进行了限制，在下一个检查点开始之前，I/O在检查点开始和结束时开启；这会在检查点进行时降低性能损耗。

服务器的检查点进程每checkpoint_segments个日志段或每 checkpoint_timeout秒就创建一个检查点，以先到为准。 缺省设置分别是 3 个段和 300 秒（5分钟）。如果自从前一个检查点以来没有WAL写入， 那么将跳过新的检查点，即使已经超过了 `checkpoint_timeout` 。 （如果使用了WAL归档并且你希望放置一个低一些的限制在多久文件归档一次上，以限制潜在的数据丢失， 你应该调整archive_timeout参数而不是检查点参数。） 我们也可以用 SQL 命令 `CHECKPOINT` 强制创建一个检查点。

减少 `checkpoint_segments` 和/或 `checkpoint_timeout` 会更频繁的创建检查点。这样就允许更快的崩溃后恢复(因为需要重做的工作更少)。不过， 我们必须在更快的恢复与更频繁的刷新脏数据页所带来的额外开销之间进行平衡。并且， 如果开启了full_page_writes(缺省开启)，那么还有其它的因素需要考虑。 为了保证数据页的一致性，在每个检查点之后的第一次数据页变化会导致对整个页面内容的日志记录。 因此，减小检查点时间间隔会导致输出到 WAL 日志中的数据量增加，从而抵销一部分使用间隔的目标， 并且无论如何都会产生更多的磁盘 I/O 操作。

检查点的开销相当高，首先是因为它需要写出所有当前脏缓冲区，其次是因为导致前面讨论过的额外后继 WAL 流量。 因此把检查点参数设置得足够高，让检查点发生的频率降低是明智的。可以通过设置 checkpoint_warning对检查点参数进行一个简单自检。如果检查点发生的间隔接近 `checkpoint_warning` 秒，那么将向服务器日志输出一条消息，建议你增加 `checkpoint_segments` 的数值。偶尔出现这样的警告并不会导致警报，但是如果出现得太频繁，那么就应该增加检查点控制参数。 如果你没有把 `checkpoint_segments` 设置得足够大，那么批量操作的时候 (比如大批的 `COPY` 传输)会导致出现大量此类警告消息。

为了避免大量的块写操作塞满I/O系统，在一段时间内，在检查点期间写脏缓冲。这个时间是由 checkpoint_completion_target控制的，作为检查点时间间隔的一小部分。 调整I/O速率以便当从检查点开始时给出的 `checkpoint_segments` 段的分数已使用完， 或已经过了给定的 `checkpoint_timeout` 秒数时完成检查点，不管时间哪个更早。 缺省值是0.5，PostgreSQL 可以在下次检查点开始之前用大约一半的时间完成检查点。 在一个正常操作时就很接近最大I/O吞吐量的操作系统上，可以增加 `checkpoint_completion_target` 以降低检查点时的I/O负载。这样做的弊端在于会延长检查点，从而影响恢复时间， 因为会保留更多的在恢复中可能用到的WAL段。尽管 `checkpoint_completion_target` 最大可以设置为1.0，最好不要设置那么大， 最大0.9，因为检查点期间的操作不仅仅包括写脏缓冲区。 设置为1.0极有可能会导致不会按时完成检查点，从而由于所需的WAL段的数目的意外变化造成性能丢失。

至少会有一个 WAL 段文件，而且通常不会超过(2 + `checkpoint_completion_target`) *`checkpoint_segments`* + *1* 或 *`checkpoint_segments`* + *wal_keep_segments* + *1*个文件。 每个段文件通常为 *16MB(*你可以在编译服务器的时候修改它*)*。你可以用这些信息来估计*WAL*需要的空间。 通常，如果一个旧日志段文件不再需要了，那么它将得到回收*(*重命名为顺序的新的可用段*)*。 如果由于短期的日志输出高峰导致了超过*3* `checkpoint_segments` + 1个段文件， 那么当系统再次回到这个限制之内的时候，不需要的段文件将被删除，而不是回收利用。

在归档恢复或待机模式时，服务器在正常操作中会定期执行类似于检查点的*restartpoints* ：服务器会强制将他的状态写入磁盘，更新 `pg_control` 文件来表明已经处理了的WAL数据不需要再次扫描，然后便会回收在 `pg_xlog` 目录下所有旧日志段文件。 重启点不能比检查点运行的更频繁，因为重启点只能在检查点记录上执行。当到达检查点记录时， 如果从最后一个重启点至少已经过去 `checkpoint_timeout` 秒，那么触发重启点。 在待机模式下，如果自从最后一个重启点已经至少 `checkpoint_segments` 个日志段重放， 也会触发重启点。

有两个常用的内部WAL函数 `XLogInsert` 和 `XLogFlush` 。 `XLogInsert` 用于向共享内存中的WAL缓冲区里添加一条新记录。 如果没有空间存放新记录，那么 `XLogInsert` 就不得不写出(向内核缓存里写) 一些填满了的WAL缓冲。我们可不想这样，因为 `XLogInsert` 用于每次数据库低层修改(比如插入记录)时都要在受影响的数据页上持有一个排它锁， 所以该操作需要越快越好；更糟糕的是，写WAL缓冲可能还会强制创建新的日志段， 它花的时间甚至更多。通常，WAL缓冲区应该由一个 `XLogFlush` 请求来写和刷新，在大部分时候它都是发生在事务提交的时候以确保事务记录被刷新到永久存储器上去了。 在那些日志输入量比较大的系统上， `XLogFlush` 请求可能不够频繁， 这样就不能避免 `XLogInsert` 进行写操作。在这样的系统上， 我们应该修改wal_buffers参数的值来增加WAL缓冲区的数量。 如果设置了 full_page_writes并且系统相当繁忙， 把 `wal_buffers` 设置得高一些将有助于在紧随每个检查点之后的时间里平滑响应时间。

commit_delay定义了在 `XLogFlush` 内请求一个锁后一组提交领导者进程将要休眠的毫秒数，组提交后面跟着排队的领导者。 这样的延迟可以允许其它的服务器进程把它们提交的记录追加到WAL缓存中， 这样就可以通过领导者的最终sync操作把所有记录刷新。如果没有打开fsync 或者当前少于commit_siblings个处于活跃事务状态的其它会话时则不会发生休眠；

这样就避免了在其它事务不会很快提交的情况下睡眠。请注意，在一些平台上， 休眠要求的分辨率是 10 毫秒，所以任何介于 1 和 10000 微秒之间的非零 `commit_delay` 设置的作用都是一样的。也要注意，在一些平台上，睡眠操作可能比参数要求的时间稍长一些。

因为 `commit_delay` 的目的是允许每个刷新操作的开销分摊给并发的提交事务 （可能是事务潜在开销），在设置可以明智的选择钱量化开销是必须的。开销越高， `commit_delay` 在一定程度上提高事务吞吐量越有效。pg_test_fsync程序可以用来测量单次WAL刷新操作使用的平均毫秒数。 这个程序报告的平均时间的一半用来在单个8kB写操作之后刷新，这个时间通常是 `commit_delay` 最有效的设置，所以当优化特定负载时，建议使用这个值作为起始点。当WAL日志存储在高时延旋转磁盘上时， 调整 `commit_delay` 是尤其有用的，即使存储媒体有非常快的sync时间也是有显著效益的， 比如有电池备用写缓存的固态硬盘或RAID阵列；但是这应该明确对代表工作负载测试。 `commit_siblings` 的更高值应该在诸如此类的情况下使用，而更小值通常对高时延媒体有帮助。注意， 总事务吞吐量太大时， `commit_delay` 的设置太高会增加事务时延是极有可能的。

当 `commit_delay` 设置为0时（缺省），仍然可能发生组提交， 但是每组将只由到达点的会话组成，在这个点他们需要刷新在前一个刷新操作（如果有） 发生时他们的提交记录。更高的客户端计数"舷梯效应"往往会发生， 所以组提交的影响会是显著的，即使 `commit_delay` 为0，并且因此明确的设置 `commit_delay` 往往没什么用处。设置 `commit_delay` 只能帮助以下情况：（1）有一些并发提交事务， （2）吞吐量通过提交率限制到某种程度； 但是对于高旋转延迟，只有两个客户端时，这个设置对增加事务吞入量是有效的（也就是， 单次提交客户端有一个兄弟事务）。

wal_sync_method参数决定PostgreSQL 如何请求操作系统内核强制将WAL更新输出到磁盘。 只要满足可靠性， 那么所有选项应该都是一样的，除 `fsync_writethrough`， 可以有时强制刷新磁盘高速缓存， 即使其他选项时不这样做。但是哪个最快则可能和平台密切相关。你可以使用 pg_test_fsync测试不同选项的速度。请注意如果你关闭了 `fsync` 的话这个参数就无关紧要了。

打开wal_debug配置参数(前提是编译PostgreSQL 的时候打开了这个支持)将导致每次 `XLogInsert` 和 `XLogFlush` WAL 调用都被记录到服务器日志。这个选项以后可能会被更通用的机制取代。

29.5. WAL 内部

WAL是自动打开的。除了要求一些磁盘空间存放WAL 日志以及一些必要的调节以外(参阅Section 29.4)，对管理员没有什么其它要求。

WAL日志存放在数据目录的 `pg_xlog` 子目录里， 它是作为一个文件段的集合存储的，通常每段 16MB （但是大小可以通过建立服务器时改变 `--with-wal-segsize` 配置选项改变）。每个段又分割成多个页，通常每页 8KB（这个大小可以通过 `--with-wal-blocksize` 配置选项改变）。日志记录头在 `access/xlog.h` 里描述； 日志内容取决于它记录的事件类型。段文件的名字是递增自然数，从 `000000010000000000000000` 开始。这些数字不能循环使用，不过要把所有可用的数字都用光也需要非常长的时间。

日志位于和主数据库文件位于不同的磁盘上会比较好。你可以通过把 `pg_xlog` 目录移动到另外一个位置(此时必须关闭服务器)，然后在原来的位置创建一个指向新位置的符号链接。

WAL的目的是确保在数据库记录被修改之前先写日志， 但是这个目的有可能被那些向内核谎报成功写入的磁盘驱动器 破坏，这时候， 它们实际上只是缓冲了数据而并未把数据存储到磁盘上。 这种情况下的电源失效仍然可能导致不可恢复的数据崩溃；管理员应该确保保存PostgreSQL的WAL日志文件的磁盘不会做这种虚假汇报。 (参阅 Section 29.1.)

在完成一个检查点并且刷新了日志文件之后，检查点的位置就保存在了 `pg_control` 文件里。因此在恢复开始的时候，后端首先读取 `pg_control` 和检查点记录； 然后通过从检查点记录里标识的日志位置开始向前扫描执行 REDO 操作。 因为数据页的所有内容都保存在检查点之后的第一个页面修改的日志里（假设 full_page_writes没有禁用），所以自检查点以来的所有变化都将被恢复到一个一致的状态。

但是为了处理 `pg_control` 可能的损坏， 我们应该支持对现存日志段的反向顺序扫描(从最新到最老)，这样才能找到最后的检查点。 这些还没有实现。 `pg_control` 很小(比一个磁盘页小)， 因此它出现只写了一部分的问题的概率几乎为零，到目前为止， 我们还没有看到不能读取 `pg_control` 自身的错误。因此， 尽管这在理论上是一个薄弱环节，但是实践中 `pg_control` 似乎并不会出现问题。

Chapter 30. 回归测试

Table of Contents

回归测试是一套复杂完整的测试，用来测试嵌入在PostgreSQL 里的 SQL 实现。它同时测试标准 SQL 操作和PostgreSQL的扩展 SQL 。

30.1. 运行测试

回归测试可以对一套已经安装好并且在运行中的服务器进行测试， 也可以对编译树里面即将安装的服务器进行测试。详细些说，有"并行" 和"串行"运行测试之分。串行模式顺序运行每个测试，而并行模式启动多个服务器进程， 并发地运行一组测试。并发测试使我们对进程内部通讯和锁的正确工作有足够的信心。

30.1.1. 对临时安装运行测试

编译之后和安装之前运行回归测试，你可以在顶级目录运行(或者进入 `src/test/regress` 子目录然后在那里运行)：

```
gmake check
```

这样将先编译几个辅助文件，比如一些用户定义的触发器函数，然后再运行测试驱动脚本。最后你会看到类似下面的东西：

```
<samp class="literal">=======================
 All 115 tests passed.
=======================</samp>
```

或者是一些关于某项测试失败的信息。先看看Section 30.2 然后再想想一个"失败"是否代表严重的错误。

因为这个测试方法运行临时的服务器，所以如果你是 root 用户， 那这个方法不能运行(服务器不能以 root 身份启动)。如果你已经以 root 身份编译了， 你就什么也干不了。这时候你应该把测试目录的权限变成某个用户可写， 然后以那个用户身份登陆，再开始测试。比如：

```
<samp class="literal">root#</samp> <kbd class="literal">chmod -R a+w src/test/regress</kb
<samp class="literal">root#</samp> <kbd class="literal">su - joeuser</kbd>
<samp class="literal">joeuser$</samp> <kbd class="literal">cd `_top-level build directory
<samp class="literal">joeuser$</samp> <kbd class="literal">gmake check</kbd>
```

这里唯一可能的"安全隐患"就是那个用户可能会背着你修改回归测试的结果。 用你的常识管理用户权限。

如果不是上面那样，安装后就可以运行测试。

如果你配置PostgreSQL安装到一个原来安装有老版本 PostgreSQL的目录里，然后在安装新版本之前执行 `gmake check` ， 那么你可能发现测试失败，因为新程序试图使用已经存在的共享库，典型的症状是抱怨未定义的符号。 如果你想在覆盖老版本之前运行测试，那么你需要

使用 `configure --disable-rpath` 进行编译。 不过，我们不建议你使用这个选项编译作为最终安装的数据库。

并发的回归测试会在你的用户 ID 下启动相当多的进程。目前，最大的并发数是 20 个并发测试脚本， 这意味着 40 个进程：一个服务器进程、每个脚本一个psql进程。因此， 如果你的系统有针对每个用户的进程数限制，那么请确保这个限制至少是 50， 否则你就可能在并发测试时看到随机出现的失败。如果你没有办法提升该限制， 那么可以通过设置 `MAX_CONNECTIONS` 参数降低并发测试程度。比如：

```
gmake MAX_CONNECTIONS=10 check
```

将运行最多不超过 10 个并发进程。

30.1.2. 对现有安装运行测试

安装后(参见 (see Chapter 15))运行测试， 像as explained in Chapter 17, 描述的那样初始化一个数据区并启动服务器，然后键入：

```
gmake installcheck
```

或者是运行一个并发测试：

```
gmake installcheck-parallel
```

该测试将与在本地主机和缺省端口号上运行的服务器进行连接，除非你用 `PGHOST` 和 `PGPORT` 环境变量设置为其它值。

源代码发布还包含给可选的过程语言和 `contrib` 模块使用的回归测试。目前， 这些测试只能用于已经安装的服务器。要给所有编译并安装的过程语言运行测试， 我们可以进入源代码树的 `src/pl` 目录然后运行：

```
gmake installcheck
```

你还可以在 `src/pl` 的任何子目录里只针对一种过程语言进行测试。 要为所有 `contrib` 模块运行测试，必须首先编译并安装 `contrib` 模块， 然后进入 `contrib` 目录运行：

```
gmake installcheck
```

你也可以在 `contrib` 的子目录里只针对一个模块运行测试。

30.1.3. 测试热备份

源代码发布还包含热备份静态行为的回归测试。测试要求一个运行的主服务器和一个运行的备用服务器， 接受使用基于文件日志传送或流复制的从主服务器改变的新WAL。这些服务器不是自动为你创建的， 设置文件也不是。请查阅所需命令和相关问题的文档的细节。

首先，在主服务器上创建一个命名为"regression"的数据库。

```
psql -h primary -c "CREATE DATABASE regression"
```

然后，在主服务器上的回归数据库运行一个准备脚本：`src/test/regress/sql/hs_primary_setup.sql`， 允许更改传送到备用，例如：

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

现在确认测试的默认连接是接受测试的备用服务器，然后从回归目录运行 `standbycheck`：

```
cd src/test/regress
gmake standbycheck
```

一些极端行为也可能在主服务器产生，使用脚本：`src/test/regress/sql/hs_primary_extremes.sql` 允许测试备用服务器的行为。

额外的自动化测试可能在以后的版本中可用。

30.1.4. 区域和编码

默认的， 对临时安装的测试使用在当前环境中定义的区域并且由 `initdb` 决定相应的数据库编码。通过设置适当的环境变量来测试不同的区域是有用的，例如：

```
gmake check LANG=C
gmake check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

由于实现原因， 设置 `LC_ALL` 并不能为此工作；所有其他区域相关的环境变量可以。

当对现有安装测试时，区域是由现有数据库集群决定的，并且不能单独为测试运行设置。

你也可以通过设置变量 `ENCODING` 明确的选择数据库编码，例如：

```
gmake check LANG=C ENCODING=EUC_JP
```

用这种方式设置数据库编码通常只在区域是C的情况下有用；否则编码自动从区域中选择，并且指定不匹配区域的编码将会导致错误。

编码可以为临时或现有安装测试设置。

30.1.5. 额外的测试

回归测试包含的一些测试文件默认是不运行的，因为它们可能是依赖于平台的或运行需要很长的时间。 你可以通过设置变量`EXTRA_TESTS`运行它们或其他额外测试文件。 例如，运行`numeric_big`测试：

```
gmake check EXTRA_TESTS=numeric_big
```

运行排序测试：

```
gmake check EXTRA_TESTS=collate.linux.utf8 LANG=en_US.utf8
```

`collate.linux.utf8`测试只在Linux/glibc平台工作， 并且只在数据库使用UTF-8编码时运行。

30.2. 测试评估

有一些正确安装并且具有完整功能的PostgreSQL 可能会在一些回归测试中"失效"，这主要是因为浮点数的形式和时区支持的问题。 目前的测试只是简单的用 `diff` 与参考系统的输出进行比较， 因而对一些细小的系统区别很敏感。当一项测试报告"失败"时， 只要检查一下预期和实际的结果，你就会发现区别并不大。当然， 我们仍然在努力维护所有我们支持的平台的准确参考文件，这样我们就可以假定所有测试都通过。

回归测试的实际输出在 `src/test/regress/results` 目录里的文件里。 测试脚本使用 `diff` 比较每个输出文件和存放在 `src/test/regress/expected` 目录里的参考输出。任何区别都存放在 `src/test/regress/regression.diffs` 里面供你检查。如果你不喜欢默认的 `diff` 选项，设置环境变量 `PG_REGRESS_DIFF_OPTS` 为 `PG_REGRESS_DIFF_OPTS='-u'` 。 你也可以自己运行 `diff` 。

如果获得一个"失败"的测试结果，但实际上输出结果是正确的， 你可以通过添加新的比较文件来抑制错误报告。参见Section 30.3获取更多细节。

30.2.1. 错误信息差别

有一些回归测试涉及到有意的非法输入值。错误信息可能会来自PostgreSQL 代码或来自主机平台系统过程。对于后者，信息可能在平台之间区别比较大， 但应该反映相似的信息。这些信息上的差别将会导致一个"失败"的回归测试， 我们可以通过检查文件发现这一点。

30.2.2. 区域差别

如果你在一台服务器上运行测试，而该服务器是用一种非 C 区域设置初始化的， 那么可能因为有排序顺序和其它类似的差别导致的失败。 回归测试套件处理这种问题的方法是提供可选的结果文件，这些文件一起处理一大堆的区域。

当使用临时安装在一个不同的区域中运行测试时，在 `make` 命令行传递适当的区域相关的环境变量，例如：

```
gmake check LANG=de_DE.utf8
```

回归测试驱动附件 `LC_ALL` ，所以它不使用那个变量选择区域。 要不使用区域，取消所有区域相关的环境变量（或设置它们为 `C` ） 或使用下列的特殊调用：

```
gmake check NO_LOCALE=1
```

当对一个现有安装运行测试时，区域设置取决于现有安装。要改变它， 通过传递适当的选项到 `initdb` ，用一个不同的区域初始化数据库集群。

通常，作为生产用的区域设置上运行回归测试是明智的，因为这将练习区域和编码相关的代码部分， 并将实际在生产中使用。取决于操作系统环境，你可能会得到错误， 但是然后你将至少知道在运行实际应用时，预期什么区域特定的行为。

30.2.3. 日期和时间差别

大多数日期和时间测试结果依赖于时区设置。参考文件是为 `PST8PDT` 时区(伯克利，加州)准备的，因而如果测试没有设置为那个时区是显然会失败的。 回归测试的驱动器把 `PGTZ` 环境变量设置为 `PST8PDT` ， 基本可以保证正确的测试。

30.2.4. 浮点数差别

有些测试涉及到对表中的数据列进行 64 位浮点数(`double precision`)计算的问题。 我们观察了涉及到计算 `double precision` 字段的数学函数的结果差别。 `float8` 和 `geometry` 测试尤其容易在不同平台， 或者甚至是不同的编译器最优化设置之间产生小差别。 这时需要肉眼对这些差别进行比较，以判断这些差别究竟有多大，我们发现是在小数点右边 10 位左右。

有些系统把负零显示为 `-0` ，而其它的只是显示 `0` 。

有些系统在 `pow()` 和 `exp()` 出错时产生的信号与目前PostgreSQL代码里期望的机制不一样。

30.2.5. 行顺序差别

你可能会看见同样的行以与预期文件的不同的顺序输出。在大多数情况下，严格说来这不算臭虫。 大多数回归测试脚本都不会迂腐到在每个 `SELECT` 中都使用 `ORDER BY` 的地步， 因此根据 SQL 规范，它们的结果行顺序并非定义得非常好的。实际上， 因为我们是在同样的数据上用同样的软件运行同样的查询，所以在所有平台上通常都获得同样的结果， 因此即使缺少 `ORDER BY` 也不算什么大问题。不过有些查询的确存在跨平台的排序问题。 在测试一台已安装的服务器的时候，排序的差别也可能因为非 C 区域设置，或者非缺省的参数设置， 比如客户自己设置的 `work_mem` 或者规划器开销参数设置受影响。

因此，如果你看到一个排序差异，应该不是什么要担心的问题(除非明确使用了 `ORDER BY`)。 不过，如果有这样的现像，请告诉我们，这样我们就可以在那条查询上加一个 `ORDER BY` 从而在以后的版本里消除这种伪"失败"。

你可能会问，为什么我们不对所有回归测试的 SELECT 进行排序以一次性消灭所有这类问题。 原因是这样做只能让回归测试用处更少，而不是更多， 因为它们会试图使用那些生成顺序结果的查询规划，而不再使用那些不排序的查询规划。

30.2.6. 堆栈深度不够

如果 `errors` 测试导致在 `select infinite_recurse()` 命令的时候服务器崩溃，这就意味着平台对进程堆栈的限制小于 max_stack_depth 参数值。 我们可以通过在更高的堆栈限制的数值上运行服务器绕开这个问题(缺省 `max_stack_depth` 建议值是 4MB)。如果你无法这么做， 那么另外一个方法是减少 `max_stack_depth` 的值。

30.2.7. "随机" 测试

`random` 测试脚本的目的是生成随机结果。在很罕见的情况下， 这会导致回归测试中的随机测试失败。键入：

```
diff results/random.out expected/random.out
```

会产生仅仅一行或几行差别。你不必担心这些，除非随机测试在重复测试中总是失败。

30.3. 平台相关的比较文件

因为一些测试天生会产生平台相关的结果，我们提供了明确指定与该平台相关的比较文件的方法。 每个回归测试都可以有针对不同平台的多个比较文件。有两个独立的机制可以用于确定究竟应该使用哪个比较文件。

第一个机制是根据特定的平台选择比较文件。通过一个映射文件 `src/test/regress/resultmap` 定义每个平台使用的比较文件。要消除某特定平台的虚假的测试"失败"， 可以先选择或创建一个结果文件的变种，然后在 `resultmap` 文件中添加一行指定映射关系即可。

映射文件里的每行都是如下形式

```
testname:output:platformpattern=comparisonfilename
```

测试名称只是特定回归测试模块的名称。输出值表明要检查哪个输出文件。 对于标准回归测试，这里总是 `out` 。这个值对应输出文件的文件扩展名。 平台名称模式是 Unix 工具 `expr` 风格的模式(一个开头带有隐含 `^` 锚符号的正则表达式)。它与 `config.guess` 打印出来的平台名匹配。 比较文件名是替换结果比较文件的基本名。

比如：一些系统把很小的浮点数解析成为零，而不是报告一个下溢的错误。 这会导致在 `float8` 回归测试中的一些差别。因此，我们提供了一个比较文件的变种 `float8-small-is-zero.out` ，它包含在这些平台上的预期结果。 要在OpenBSD平台上消除这些虚假的"错误"信息， 可以在 `resultmap` 中包含：

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

它将在那些 `config.guess` 的输出匹配 `i.86-.*-openbsd` 的任何机器上触发。在 `resultmap` 里的其它行同样为其它合适的平台选取相应的比较文件变种。

第二个选择比较文件的机制更加自动化：它简单的在多个比较文件中使用"最佳匹配"。 回归测试的驱动脚本同时考虑标准比较文件、 `_testname_` .out、 以及名为 `_testname_` `_*digit* `.out(` *digit* `是` 0 `-` 9 `的任意一个)的变种文件。如果其中之一完全符合就认为测试通过， `resultmap `包含某个特定测试的项， 那么基准` *testname* `就是` resultmap`中给出的替换名。

例如， 对于 `char` 测试，比较文件 `char.out` 包含使用 `C` 和 `POSIX` 区域设置的结果，而 `char_1.out` 文件则包含使用其它区域设置的结果。

最佳匹配机制主要目的是用于匹配区域相关的测试结果，但也可以用于仅凭平台名称难以预计测试结果的场合。 这个机制的一个缺点是测试脚本无法确定当前环境下究竟哪个变种是"确切"的， 它只能选择最贴近的变种。因此最好将这个机制仅仅用于多个变种在所有上下文环境中都可以被认为是等价的场合。

30.4. 测试覆盖率检查

PostgreSQL源代码可以编译有覆盖测试设备， 所以检查回归测试包含哪部分代码或任何其他测试组件运行代码是可能的。 使用GCC编译并且需要 `gcov` 和 `lcov` 程序是目前支持的。

一个典型的工作流应该像这样：

```
./configure --enable-coverage ... OTHER OPTIONS ...
gmake
gmake check # or other test suite
gmake coverage-html
```

然后你的HTML浏览器跳转到 `coverage/index.html` 。 `gmake` 命令在子目录中也能工作。

要重置测试运行之间的执行计数，运行：

```
gmake coverage-clean
```

IV. 客户端接口

这部分描述和 PostgreSQL 一起发布的客户端编程接口。 这里的每一章都可以独立阅读。 请注意还有许多用于客户端程序的编程接口是独立发布的， 它们包含自己的文档（Appendix H 列出了一些比较流行的）。 这部分的读者应该熟悉使用 SQL 命令操作和查询数据库（参阅 Part II）， 并且当然也得熟悉接口使用的编程语言。

Table of Contents

Chapter 31. libpq - C 库

Table of Contents

libpq是PostgreSQL的 C应用程序接口。libpq 是一套允许客户程序向PostgreSQL 服务器服务进程发送查询并且获得查询返回的库函数。

libpq同时也是其他几个PostgreSQL 应用接口下面的引擎，包括 C++，Perl，Python，Tcl 和 ECPG。 所以如果你使用这些软件包，libpq 某些方面的特性会对你非常重要。特别是Section 31.14， Section 31.15和Section 31.18 描述了任何使用libpq的应用的用户可见的行为。

本章末尾有三个小程序显示如何利用libpq书写程序。 （Section 31.21）在源代码发布的 `src/test/examples` 目录里面有几个完整的libpq应用的例子。

使用libpq的前端程序必须包括头文件 `libpq-fe.h` 并且必须与libpq库链接。

31.1. 数据库连接控制函数

下面的函数处理与PostgreSQL服务器联接的事情。 一个应用程序一次可以与多个服务器建立联接。 （这么做的原因之一是访问多于一个数据库。） 每个连接都是用一个从函数 `PQconnectdb` 、 `PQconnectdbParams` 或 `PQsetdbLogin` 获得的 `PGconn` 对象表示。 注意，这些函数总是返回一个非空的对象指针，除非存储器少得连个 `PGconn` 对象都分配不出来。在把查询发送给连接对象之前， 可以调用 `PQstatus` 函数来检查一下返回值看看连接是否成功。

> **Warning**
>
> 在Unix上，用打开的libpq连接分支化一个过程会导致不可预知的结果， 因为父进程和子进程共享同一个套接字和操作系统资源。因为这个原因， 不建议这样的使用，尽管从子进程中执行 `exec` 加载一个新的可执行文件是安全的。

> **Note:** 在Windows上，如果一个数据库连接重复的启动和关闭，有一个方式提高性能。 内部的，libpq为连接启动和关闭分别调用 `WSAStartup()` 和 `WSACleanup()` 。 `WSAStartup()` 增加一个内部Windows库引用计数，而 `WSACleanup()` 减少一个。当引用计数是一时，调用 `WSACleanup()` 释放所有资源和所有DLL是空载的。 这是一个昂贵的操作。为了避免它，一个应用可以手动调用 `WSACleanup()` ， 这样在最后一个数据库连接关闭时，资源将不会被释放。

`PQconnectdbParams`

与数据库服务器建立一个新的连接。

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

这个函数用从两个 `NULL` 结束的数组中来的参数打开一个新的数据库连接。 第一个， `keywords` ，定义为一个字符串的数组，每个都成为一个关键字。 第二个， `values` ，给每个关键字一个值。与下面的 `PQsetdbLogin` 不同的是，我们可以不必更换函数签名（名字）就可以扩展参数集， 所以我们建议应用程序中使用这个函数（或者它的类似的非阻塞变种 `PQconnectStartParams` 和 `PQconnectPoll` ）。

目前公认的参数关键字在Section 31.1.2中列出。

当 `expand_dbname` 是非零的时，允许将 `dbname` 的关键字值看做一个连接字符串。可能的格式的更详细信息在Section 31.1.1 中显示。

传入的参数可以为空，表明使用所有缺省的参数，或者可以包含一个或更多个参数设置。 它们的长度应该匹配。处理将会在 `keywords` 数组的最后一个非 `NULL` 元素停止。

如果没有指定任何参数，则使用对应的环境变量（参阅Section 31.14）。 如果环境变量也没有设置，则使用表示的内建缺省。

通常，关键字是从这些数组的开始以索引的顺序处理的。这样的影响是，当关键字重复时，获得最后处理的值。因此，通过小心的放置 `dbname` 关键字， 有可能决定哪个被 `conninfo` 字符串覆盖，哪个不被覆盖。

`PQconnectdb`

与数据库服务器建立一个新的连接。

```
PGconn *PQconnectdb(const char *conninfo);
```

这个函数用从一个字符串 `conninfo` 来的参数与数据库打开一个新的联接。

传入的参数可以为空，表明使用所有缺省的参数，或者可以包含一个或更多个用空白间隔的参数设置， 或者它可以包含一个URI。参阅Section 31.1.1获取细节。

`PQsetdbLogin`

与数据库服务器建立一个新的连接。

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

这个函数是 `PQconnectdb` 前身，它有固定个数的参数。它有相同的功能， 只是在调用中那些它缺少的参数总是用缺省值。如果要给任意的固定参数设置缺省值， 那么写一个 `NULL` 或者一个空字串给它们。

如果 `dbName` 包含一个 `=` 符或者有一个有效的连接 URI前缀，它被看做一个 `conninfo` 字符串， 就和它已经被传递到 `PQconnectdb` 中完全一样， 然后剩余的参数就像为 `PQconnectdbParams` 指定的那样应用。

`PQsetdb`

与数据库服务器建立一个新的连接。

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

这是一个调用 `PQsetdbLogin` 的宏，只是 `login` 和 `pwd` 参数是空指针。提供这个函数是为了与非常老版本的程序兼容。

`PQconnectStartParams``PQconnectStart` `PQconnectPoll`

与数据库服务器建立一次非阻塞的联接。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

这三个函数用于打开一个与数据库服务器之间的非阻塞的联接： 你的应用的执行线程在执行它的时候不会因远端的 I/O 而阻塞。 这个方法的要点是等待 I/O 结束可以发生在应用的主循环里， 而不是在 `PQconnectdbParams` 或 `PQconnectdb` 里， 这样应用可以把这件事与其它操作并发起来一起执行。

对于 `PQconnectStartParams` ，数据库联接是用从 `keywords` 和 `values` 数组中取得的参数进行的，并且是使用 `expand_dbname` 控制的，就像上面 `PQconnectdbParams` 里描述的一样。

对于 `PQconnectStart` ，数据库联接是用从 `conninfo` 字符串里取得的参数进行的，这个字符串的格式与上面 `PQconnectdb` 里描述的一样。

`PQconnectStartParams` 、 `PQconnectStart` 和 `PQconnectPoll` 都不会阻塞（进程），不过有一些条件：

- 必须正确提供 `hostaddr` 和 `host` 参数以确保不会发生正向或者反向的名字查找。 参阅Section 31.1.2里的这些参数的文档获取细节。
- 如果你调用了 `PQtrace` ，确保你跟踪进入的流对象不会阻塞。
- 你必须在调用 `PQconnectPoll` 之前确保 socket 处于正确的状态， 像下面描述的那样。

注意： `PQconnectStartParams` 的使用类似于下面显示的 `PQconnectStart` 。

要开始一次非阻塞连接请求，调用 `conn = PQconnectStart("``_connection_info_string_ ")`。 如果 `conn` 是空，表明libpq无法分配一个新的 `PGconn` 结构。 否则，返回一个有效的 `PGconn` 指针（尽管还不一定代表一个与数据库有效联接）。 `PQconnectStart` 一返回，调用 `status = PQstatus(conn)` 。 如果 `status` 等于 `CONNECTION_BAD` ， `PQconnectStart` 失败。

如果 `PQconnectStart` 成功了，下一个阶段是轮询libpq， 这样它就可以继续连接序列动作。使用 `PQsocket(conn)` 获取数据库链接下层的套接字描述符。像这样循环：如果 `PQconnectPoll(conn)` 的最后一个返回是 `PGRES_POLLING_READING` ，那么就等到套接字准备好被读取了的时候 （就像系统函数 `select()` ， `poll()` ，或者类似的系统调用声明的那样）。 然后再次调用 `PQconnectPoll(conn)` 。反过来，如果 `PQconnectPoll(conn)` 最后返回 `PGRES_POLLING_WRITING` ，那么就等到套接字准备好可以写了， 然后再次调用 `PQconnectPoll(conn)` 。如果你还没调用 `PQconnectPoll` ， 比如，刚刚调用

完 `PQconnectStart` ，那么按照它刚返回 `PGRES_POLLING_WRITING` 的原则行动。继续这个循环直到 `PQconnectPoll(conn)` 返回 `PGRES_POLLING_FAILED` ， 表明连接过程失败，或者 `PGRES_POLLING_OK` ，表明连接成功建立。

在连接的任意时刻，我们都可以通过调用 `PQstatus` 来检查联接的状态。 如果这是 `CONNECTION_BAD` ，那么联接过程失败；如果是 `CONNECTION_OK` ， 那么联接已经做好。这两种状态同样也可以从上面的 `PQconnectPoll` 的返回值里检测到。 其他状态可能（也只能）在一次异步联接过程中发生。这些标识连接过程的当前状态， 因而可能对给用户提供反馈有帮助。这些状态可能包括：

`CONNECTION_STARTED`

等待进行连接。

`CONNECTION_MADE`

连接成功；等待发送。

`CONNECTION_AWAITING_RESPONSE`

等待来自服务器的响应。

`CONNECTION_AUTH_OK`

已收到认证；等待后端启动结束。

`CONNECTION_SSL_STARTUP`

协商 SSL 加密。

`CONNECTION_SETENV`

协商环境驱动的参数设置。

注意，尽管这些常量将保持下去（为了维持兼容性）， 应用决不应该依赖于这些常量以某种特定顺序出现， 或者是根本不应依赖于这些常量，或者是不应该依赖于这些状态总是某个文档声明的值。 一个应用可能像下面这样：

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Connecting...";
            break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
            feedback = "Connecting...";
}
```

在使用 `PQconnectPoll` 的时候，连接参数 `connect_timeout` 将被忽略；判断是否超时是应用的责任。否则，后面跟着一个 `PQconnectPoll` 循环的 `PQconnectStart` 等效于 `PQconnectdb` 。

要注意如果 `PQconnectStart` 返回一个非空的指针，你必须在使用完它（指针） 之后调用 `PQfinish` ，以处理那些结构和所有相关的存储块。 甚至是在连接尝试失败或放弃时也要这样处理。

`PQconndefaults`

返回缺省的联接选项。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* 选项的键字 */
    char   *envvar;    /* 退守的环境变量名 */
    char   *compiled;  /*  退守的编译时缺省值*/
    char   *val;       /* 选项的当前值，或者 NULL */
    char   *label;     /* 连接对话里字段的标识 */
    char   *dispchar;  /* 在连接对话里为此字段显示的字符。
                          数值有：
                          ""        原样现实输入的数值
                          "*"       口令字段 — 隐藏数值
                          "D"       调试选项 — 缺省的时候不显示 */
    int     dispsize;  /*  对话中字段的以字符计的大小 */
} PQconninfoOption;
```

返回一个连接选项数组。可以用于获取所有可能的 `PQconnectdb` 选项和它们的当前缺省值。返回值指向一个 `PQconninfoOption` 结构的数组，该数组以一个有 NULL `keyword` 指针的条目结束。 如果无法分配内存，则返回空指针。注意当前缺省值（ `val` 域） 将依赖于环境变量和其他环境。调用者必须把连接选项当作只读对待。

在处理完选项数组后，把数组交给 `PQconninfoFree` 释放。 如果没有这么做，每次调用 `PQconndefaults` 都会有一小部分内存泄漏。

`PQconninfo`

返回活的连接使用的连接选项。

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

返回一个连接选项数组。可以用于获取所有可能的 `PQconnectdb` 选项和用于连接到服务器的值。返回值指向一个 `PQconninfoOption` 结构的数组，该数组以一个有 NULL `keyword` 指针的条目结束。 以上所有 `PQconndefaults` 的注意事项也应用到 `PQconninfo` 的结果。

`PQconninfoParse`

从提供的连接字符串中返回解析的连接选项。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

解析连接字符串并作为数组返回结果选项；或者如果连接字符串有问题返回 `NULL` 。 这个函数可以用来在提供的连接字符串中提取 `PQconnectdb` 选项。 返回值指向一个 `PQconninfoOption` 结构的数组， 该数组以一个有 NULL `keyword` 指针的条目结束。

所有合法选项将在结果数组中显示，但是 `PQconninfoOption` 的任何没有在连接字符串中出现的选项将把 `val` 设置为 `NULL` ； 缺省值不插入。

如果 `errmsg` 是非 `NULL` 的，那么 `*errmsg` 在成功时设置为 `NULL` ， 否则是 `malloc` 的解释问题的错误字符串。（ `*errmsg` 设置为 `NULL` 并且函数返回 `NULL` 是可能的；这表示一个内存溢出条件。）

在处理完选项数组后，把数组交给 `PQconninfoFree` 释放。 如果没有这么做，每次调用 `PQconndefaults` 都会有一小部分内存泄漏。 相反的，如果错误发生了并且 `errmsg` 非 `NULL` ， 确保使用 `PQfreemem` 释放错误字符串。

`PQfinish`

关闭与服务器的连接。同时释放被 `PGconn` 对象使用的存储器。

```
void PQfinish(PGconn *conn);
```

注意，即使与服务器的连接尝试失败（可由 `PQstatus` 判断）， 应用也要调用 `PQfinish` 释放被 `PGconn` 对象使用的存储器。不应该在调用 `PQfinish` 后再使用 `PGconn` 指针。

`PQreset`

重置与服务器的通讯端口。

```
void PQreset(PGconn *conn);
```

此函数将关闭与服务器的连接并且试图与同一个服务器重建新的连接， 使用所有前面使用过的参数。这在失去工作连接后进行故障恢复时很有用。

`PQresetStart` `PQresetPoll`

以非阻塞模式重置与服务器的通讯端口。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

此函数将关闭与服务器的连接并且试图与同一个服务器重建新的连接，使用所有前面使用过的参数。 这在失去工作连接后进行故障恢复时很有用。它们和上面的 `PQreset` 的区别是它们工作在非阻塞模式。这些函数的使用有与上面 `PQconnectStartParams` 、 `PQconnectStart` 和 `PQconnectPoll` 一样的限制。

要发起一次连接重置，调用 `PQresetStart` 。如果它返回 0， 那么重置失败。如果返回 1，用与使用 `PQresetPoll` 建立连接的同样的方法使用 `PQresetPoll` 重置连接。

`PQpingParams`

`PQpingParams` 报告服务器的状态。它接受和 `PQconnectdbParams` 一样的连接参数，在下面描述。不需要应用正确的用户名、密码或数据库名的值获取服务器状态；不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

该函数返回下列的值之一：

`PQPING_OK`

服务器正在运行并且似乎接受了连接。

`PQPING_REJECT`

服务器正在运行，但是在一个不允许连接的状态（启动、关闭或崩溃恢复）。

`PQPING_NO_RESPONSE`

联系不上服务器。这可能表明服务器没有运行，或者给出的连接参数有什么错误 （例如，错误的端口号），或者网络连接有问题（例如，防火墙阻塞连接请求）。

`PQPING_NO_ATTEMPT`

没有尝试连接到服务器，因为提供的参数明显的不正确或者有一些客户端侧的问题（例如，内存溢出）。

`PQping`

`PQping` 报告服务器的状态。它接受和 `PQconnectdb` 一样的连接参数，在下面描述。不需要应用正确的用户名、密码或数据库名的值获取服务器状态； 不过，如果提供了不正确的值，服务器将记录一次失败的连接尝试。

```
PGPing PQping(const char *conninfo);
```

返回值和 `PQpingParams` 的相同。

31.1.1. 连接字符串

几个libpq函数分析用户指定的字符串以获取连接参数。 这些字符串有两个可接受的格式：纯 `keyword = value` 字符串和 RFC 3986 URIs。

31.1.1.1. 关键字/值连接字符串

在第一中格式中，每个参数以 `keyword = value` 的形式设置。 等号周围的空白是可选的。要写一个空值或者一个包含空白的值，你可以用一对单引号包围它们， 例如， `keyword = 'a value'` 。数值内部的单引号和反斜杠必须用一个反斜杠逃逸， 比如， `\'` 或 `\\` 。

示例：

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

可识别的参数关键字在Section 31.1.2中列出。

31.1.1.2. 连接URI

连接URI的通用格式是：

```
postgresql://[user[:password]@][netloc][:port][/dbname][?param1=value1&...]
```

URI模式标识符可以是 `postgresql://` 或 `postgres://` 。URI的每个部分都是可选的。 下列示例举例说明了有效的URI语法使用：

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
```

URI的层次部分的组件也可以作为参数给出。例如：

```
postgresql:///mydb?host=localhost&port=5433
```

百分号可以用在URI的任何部分来包含特殊含义的符号。

忽略任何不对应于在Section 31.1.2列出的关键字的连接参数， 并将关于它们的警告消息发送到 `stderr` 。

为了提高JDBC连接URI的兼容性，参数 `ssl=true` 的实例被翻译成 `sslmode=require` 。

主机部分是主机名或者IP地址。要指定一个IPv6主机地址，将它包含在方括号中：

```
postgresql://[2001:db8::1234]/database
```

主机部分解释为参数host的描述。特别的， 如果主机部分为空或者以斜线开头，那么选择一个Unix域套接字连接，否则初始化一个TCP/IP连接。 不过要注意，斜线是URI分层部分的一个保留字符。所以，要指定一个非标准Unix域套接字路径， 要么在URI中省略主机声明并指定主机为一个参数，要么在URI的主机部分添加百分号：

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

31.1.2. 参数关键字

目前可识别的参数键字是：

`host`

要联接的主机名。如果主机名以斜杠开头， 则它声明使用 Unix 域套接字通讯而不是 TCP/IP 通讯；该值就是套接字文件所存储的目录。 如果没有声明 `host`， 那么缺省时是与位于 `/tmp` 目录 （或者制作PostgreSQL的时候声明的套接字目录）里面的 Unix-域套接字连接。 在没有 Unix 域套接字的机器上，缺省是与 `localhost` 连接。

`hostaddr`

与之连接的主机的 IP 地址。这个应该是标准的IPv4 地址格式，比如， `172.28.40.9` 。 如果你的机器支持 IPv6，那么你也可以使用 IPv6 的地址。如果声明了一个非空的字符串， 那么使用 TCP/IP 通讯机制。

使用 `hostaddr` 取代 `host` 可以让应用避免一次主机名查找， 这一点对于那些有时间约束的应用来说可能是非常重要的。不过，Kerberos、GSSAPI 或SSPI认证方法和 `verify-full` SSL 证书验证要求主机（host）名。 因此，应用下面的规则：

- 如果声明了不带 `hostaddr` 的 `host` 那么就强制进行主机名查找。
- 如果声明中没有 `host` ， `hostaddr` 的值给出服务器网络地址； 如果认证方法要求主机名，那么连接尝试将失败。
- 如果同时声明了 `host` 和 `hostaddr` ，那么 `hostaddr` 的值作为服务器网络地址。 `host` 的值将被忽略，除非认证方法需要它， 在这种情况下它将被用作主机名。

要注意如果 `host` 不是网络地址 `hostaddr` 处的服务器名， 那么认证很有可能失败。同样，在 `~/.pgpass` （参阅Section 31.15） 中是使用 `host` 而不是 `hostaddr` 来标识连接。

如果主机名（host）和主机地址都没有，那么libpq 将使用一个本地的 Unix 域套接字进行连接；或者是在没有 Unix 域套接字的机器上， 它将尝试与 `localhost` 连接。

`port`

主机服务器的端口号，或者在 Unix 域套接字联接时的套接字扩展文件名。

`dbname`

数据库名。缺省和用户名相同。在某些情况下，为扩展的格式检查值；参阅Section 31.1.1获取更多信息。

`user`

要连接的PostgreSQL用户名。 缺省是与运行该应用的用户操作系统名同名的用户。

`password`

如果服务器要求口令认证，所用的口令。

`connect_timeout`

连接的最大等待时间，以秒计（用十进制整数字串书写）。零或者不声明表示无穷。 我们不建议把连接超时的值设置得小于 2 秒。

`client_encoding`

为这个连接设置 `client_encoding` 配置参数。 除了对应的服务器选项接受的值，你可以使用 `auto` 从客户端中的当前环境中确定正确的编码（Unix系统上是 `LC_CTYPE` 环境变量）。

`options`

添加命令行选项以在运行时发送到服务器。例如，设置为 `-c geqo=off` 设置 `geqo` 参数的会话的值为 `off` 。关于可用选项的详细讨论， 请查阅Chapter 18。

`application_name`

为application_name配置参数指定一个值。

`fallback_application_name`

为application_name配置参数指定一个回退值。 如果没有通过连接参数或 `PGAPPNAME` 环境变量给定 `application_name` 值，那么将使用这个值。 在想要设置缺省应用名但是允许用户重写的通用实用程序中指定一个回退名是有用的。

`keepalives`

控制客户端侧的TCP保持激活是否使用。缺省值是1，意思为打开，但是如果不想要保持激活， 你可以更改为0，意思为关闭。通过Unix域套接字做的连接忽略这个参数。

`keepalives_idle`

在TCP应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。 0值表示使用系统缺省。通过Unix域套接字做的连接或者如果禁用了保持激活则忽略这个参数。 只有在 `TCP_KEEPIDLE` 和 `TCP_KEEPALIVE` 套接字选项可用的系统上支持这个参数， 在Windows上还是在其他系统上是没什么影响的。

`keepalives_interval`

在TCP保持激活信息没有被应该传播的服务器承认之后，控制秒数。0值表示使用系统缺省。通过Unix域套接字做的连接或者如果禁用了保持激活则忽略这个参数。 只有在 `TCP_KEEPINTVL` 套接字选项可用的系统上支持这个参数， 在Windows上还是在其他系统上是没什么影响的。

`keepalives_count`

在认为客户端到服务器的连接死亡之前，控制可以丢失的TCP保持激活的数量。0值表示使用系统缺省。 通过Unix域套接字做的连接或者如果禁用了保持激活则忽略这个参数。 只有在 `TCP_KEEPINTVL` 套接字选项可用的系统上支持这个参数， 在Windows上还是在其他系统上是没什么影响的。

`tty`

忽略（以前，这个选项声明服务器日志的输出方向）。

`sslmode`

这个选项决定是否需要和服务器协商一个SSL TCP/IP连接， 以及以什么样的安全优先级与服务器进行SSL TCP/IP连接。 这里有六个模式：

`disable`

只进行一个非SSL连接

`allow`

首先尝试一个非SSL连接；如果失败， 尝试一个SSL连接

`prefer` (default)

首先尝试SSL连接；如果失败， 尝试一个非SSL连接

`require`

尝试一个SSL连接。如果有根CA文件，则按照指定了 `verify-ca` 的相同方式验证该证书

`verify-ca`

只尝试一个SSL连接，并核实服务器证书是由一个受信任的认证中心(CA)发布的

`verify-full`

只尝试一个SSL连接，核实服务器证书是由受信任的CA发布的， 并且该服务器主机名匹配证书中的服务器主机名。

参阅Section 31.18获取这些选项工作的详细描述。

Unix域套接字通信忽略 `sslmode` 。如果PostgreSQL 编译时没有打开 SSL 支持，那么使用选项 `require` 、 `verify-ca` 或 `verify-full` 将导致一个错误，而选项 `allow` 和 `prefer` 将被接受，但是libpq实际上不会企图进行SSL连接。

`requiressl`

这个选项因为有了`sslmode`设置之后已经废弃了。

如果设为1，则要求与服务器进行SSL联接（等效于`sslmode` `require`）。如果服务器不支持SSL，那么libpq 将马上拒绝联接。设置为0（缺省），与服务器进行协商连接类型（等效于`sslmode` `prefer`）。这个选项只有在编译PostgreSQL时打开了 SSL 支持才有效。

`sslcompression`

如果设置为1（缺省），通过SSL连接进行的数据发送将被压缩（这要求OpenSSL 版本0.9.8或更高）。如果设置为0，将禁用压缩（这需要OpenSSL 1.0.0或更高）。 如果连接没有通过SSL进行，或者如果使用的OpenSSL版本不支持它，则忽略该参数。

压缩使用CPU时间，但是如果网络是瓶颈，那么可以提高吞吐量。 如果CPU性能是限制因素，那么禁用压缩可以提高响应时间和吞吐量。

`sslcert`

这个参数指定客户端SSL认证的文件名，替换缺省的`~/.postgresql/postgresql.crt`。 如果没有做SSL连接，则忽略这个参数。

`sslkey`

这个参数指定客户端使用的秘钥的位置。也可以指定一个用来替换缺省`~/.postgresql/postgresql.key`的文件名，或者指定一个从外部 "引擎"获取的键（引擎是OpenSSL可加载模块）。 一个外部引擎声明应该包括一个由冒号分隔的引擎名字和特定于引擎的键标识符。 如果没有做SSL连接则忽略这个参数。

`sslrootcert`

这个参数声明一个包含SSL认证授权(CA)证书的文件名。 如果该文件存在，那么将要验证的服务器的证书将由这些授权之一签署。 缺省是`~/.postgresql/root.crt`。

`sslcrl`

这个参数声明SSL证书撤销列表(CRL)的文件名。在这个文件中列出的证书， 如果该文件存在，将在尝试认证服务器的证书时被拒绝。缺省是`~/.postgresql/root.crl`。

`requirepeer`

这个参数声明服务器的操作系统用户名，例如`requirepeer=postgres`。 当制作一个Unix域套接字连接时，如果设置了该参数，那么在连接的开始， 客户端检查服务器进程是否运行在指定的用户名之下；如果不是，则连接带有错误退出。 这个参数可以用来提供服务器认证，类似于在TCP/IP连接上可用SSL证书。 （请注意，如果Unix域套接字在`/tmp`中或另一个公开可写位置， 那么任意用户都可以在这里启动一个服务器监听。 使用这个参数确保你连接到一个受信任的用户运行的服务器。） 这个选项只有在实现了`peer`认证方法的平台上支持；参阅 Section 19.3.7。

`krbsrvname`

使用Kerberos 5或GSSAPI认证时使用的Kerberos服务名。 这个名字必须和服务器给Kerberos认证配置的服务名相同，才能认证成功。 （又见Section 19.3.5和Section 19.3.3。）

`gsslib`

为GSSAPI认证使用的GSS库。只在Windows上使用。设置为 `gssapi` 强迫libpq为认证使用GSSAPI库而不是缺省的SSPI。

`service`

用于额外参数的服务名。它在 `pg_service.conf` 里面声明一个服务名， 这个配置文件保存额外的连接参数。这样就允许应用只声明一个服务名， 而连接参数就可以在一个地方维护了。参阅Section 31.16。

31.2. 连接状态函数

这些函数可以用于询问现存数据库连接对象的状态。

> **Tip:** libpq应用程序员应该仔细维护 `PGconn` 结构。 使用下面的访问函数来获取 `PGconn` 的内容。不建议使用 `libpq-int.h` 引用 `PGconn` 内部的字段，因为这些字段在今后可能被改变。

下面的函数返回连接建立时的参数值。这些参数在 `PGconn` 对象的生命期期间是固定的。

`PQdb`

返回连接的数据库名。

```
char *PQdb(const PGconn *conn);
```

`PQuser`

返回连接的用户名。

```
char *PQuser(const PGconn *conn);
```

`PQpass`

返回连接的口令。

```
char *PQpass(const PGconn *conn);
```

`PQhost`

返回连接的服务器主机名。

```
char *PQhost(const PGconn *conn);
```

`PQport`

返回连接的端口号。

```
char *PQport(const PGconn *conn);
```

`PQtty`

返回连接的调试控制台TTY。（这个已经过时了， 因为服务器不再注意TTY设置，这个函数的存在是为了向下兼容。）

```
char *PQtty(const PGconn *conn);
```

`PQoptions`

返回连接请求中传递的命令行选项。

```
char *PQoptions(const PGconn *conn);
```

下面的函数返回那些在对 `PGconn` 对象进行操作的过程中可能变化的状态数据。

`PQstatus`

返回连接的状态。

```
ConnStatusType PQstatus(const PGconn *conn);
```

这个状态可以是一系列值之一。不过，我们在一个异步连接过程外面只能看到其中的两个：`CONNECTION_OK` 和 `CONNECTION_BAD` 。成功连接到数据库返回状态 `CONNECTION_OK` 。失败的连接尝试用状态 `CONNECTION_BAD` 标识。通常，一个OK状态将保持到 `PQfinish` ，但是一个通讯失败可能会导致状态过早的改变为 `CONNECTION_BAD` 。这时应用可以试着调用 `PQreset` 来恢复。

参阅 `PQconnectStartParams` 、 `PQconnectStart` 和 `PQconnectPoll` 条目看看可能出现的其他状态码。

`PQtransactionStatus`

返回服务器的当前事务内状态。

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

状态可以是 `PQTRANS_IDLE` （当前空闲）， `PQTRANS_ACTIVE` （正在处理一个命令）， `PQTRANS_INTRANS` （空闲，在一个合法的事务块内）， 或者 `PQTRANS_INERROR` （空闲，在一个失败的事务块内）。如果连接有问题， 则返回 `PQTRANS_UNKNOWN` 。只有在一个查询发送给了服务器并且还没有完成的时候才返回 `PQTRANS_ACTIVE` 。

> **Caution**
>
> 当使用参数 `autocommit` 设置为关闭的PostgreSQL 7.3服务器时， `PQtransactionStatus` 将给出不正确的结果。服务器端自动提交特性已经废弃了， 并且在后来的服务器版本中不再存在。

`PQparameterStatus`

查找服务器的一个当前参数设置。

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

有些参数值在建立连接或者它们的值改变的时候会由服务器自动报告。 `PQparameterStatus` 可以用查询这些设置。如果参数已知， 那么它返回当前值，否则返回 `NULL` 。

当前版本报告的参数有 `server_version` ， `server_encoding` ， `client_encoding` ， `application_name` ， `is_superuser` ， `session_authorization` ， `DateStyle` ， `IntervalStyle` ， `TimeZone` ， `integer_datetimes` 和 `standard_conforming_strings` 。 （8.0之前的版本不报告 `server_encoding` ， `TimeZone` 和 `integer_datetimes` ；8.1之前的版本不报告 `standard_conforming_strings` ；8.4之前的版本不报告 `IntervalStyle` ；9.0之前的版本不报告 `application_name` 。） 请注意 `server_version` ， `server_encoding` 和 `integer_datetimes` 不能再启动后修改。

协议版本3.0之前的服务器不会报告参数设置，但是libpq 里包含一些逻辑用于获取 `server_version` 和 `client_encoding` 的数值。 我们鼓励应用里面使用 `PQparameterStatus` ，而不是使用*ad hoc* 代码来检测这些值。 （不过要注意，在3.0之前的连接协议里，启动后通过 `SET` 改变了 `client_encoding` 将不会被 `PQparameterStatus` 反映出来。） 对于 `server_version` ，又见 `PQserverVersion` ， 它返回数值形式，更容易进行比较。

如果没有为 `standard_conforming_strings` 报告数值，应用可以假设它是 `off` ， 也就是说，在字符串文本里，把反斜杠当做逃逸。同样，如果出现了这个参数， 就可以当作一个指示，表示接受逃逸字符串(`E'...'`)的语法。

尽管返回的指针声明为 `const` ，它实际上指向一个和 `PGconn` 结构关联的可变存储区。因此假设这个指针跨查询保持有效是不明智的。

`PQprotocolVersion`

查询使用的前/后端协议。

```
int PQprotocolVersion(const PGconn *conn);
```

应用可能希望使用这个函数来判断某些特性是否被支持。目前，可能的数值是2 （2.0协议），3（3.0协议）或0（连接错误）。在连接启动完成之后，这个数值将不会改变， 但是在连接重置的过程中，理论上是可能改变的。在与PostgreSQL 7.4 或更高版本沟通时，通常使用3.0协议；7.4以前的服务器只支持协议2.0。 （协议1.0过时了，不被libpq支持。）

`PQserverVersion`

返回一个整数，代表后端版本。

```
int PQserverVersion(const PGconn *conn);
```

应用可以使用这个函数判断它们连接的数据库服务器的版本。数字是通过把主、 次及版本号转换成两位十进制数并且把它们连接在一起组成的。例如， 版本8.1.5将被返回80105，版本8.2将被返回80200（前导零没有显示）。 如果连接失败，则返回零。

`PQerrorMessage`

返回连接中操作产生的最近的错误信息。

```
char *PQerrorMessage(const PGconn *conn);
```

几乎所有libpq函数在失败时都会为 `PQerrorMessage` 设置一个信息。注意，libpq的传统是，一个非空的 `PQerrorMessage` 结果会由多行组成，并且将包含一个结尾的新行。 调用者不应该直接释放结果。结果的释放是在将 `PGconn` 句柄传递给 `PQfinish` 的时候自动进行的。我们不能假设在不同的 `PGconn` 结构操作中，结果字串都是一样的。

`PQsocket`

获取与服务器连接的套接字的文件描述符编号。一个有效的描述符应该是大于或等于0； 结果为-1表示当前没有与服务器的连接打开。（在正常的操作中，这个结果不会改变， 但是可能在连接启动或者重置的过程中变化。）

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

返回后端进程处理此连接的进程号ID (PID)

```
int PQbackendPID(const PGconn *conn);
```

这个后端PID在调试和对比 `NOTIFY` 信息（包括发出通知的后端进程的PID）时很有用。 注意该PID属于运行数据库服务器主机的进程， 而不是本地主机！

`PQconnectionNeedsPassword`

如果连接的认证方法需要一个密码则返回true (1)，但是没有可用的。 如果没有则返回false (0)。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

此功能可用于连接尝试失败后决定是否提示用户输入密码。

`PQconnectionUsedPassword`

如果连接的认证方法使用密码则返回true (1)。否则返回false (0)。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

此功能可应用于失败或成功连接后尝试检测服务器是否要求密码。

`PQgetssl`

返回连接中使用的SSL结构，或者没有使用SSL则返回null。

```
void *PQgetssl(const PGconn *conn);
```

这个结构可以用于核实加密级别，检查服务器认证等信息。参考OpenSSL 文档获取关于这个结构的更多信息。

实际返回值的类型是`SSL *`，而`SSL`的类型是由 OpenSSL库定义的，但是没有用这种方式声明， 以避免请求OpenSSL头文件。要使用这个函数， 可以使用下面的代码行：

```
#include &lt;libpq-fe.h&gt;
#include &lt;openssl/ssl.h&gt;

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQgetssl(dbconn);
    if (ssl)
    {
        /* use OpenSSL functions to access ssl */
    }
```

31.3. 命令执行函数

一旦与数据库服务器的连接成功建立，便可以使用这里描述的函数执行SQL查询和命令。

31.3.1. 主函数

`PQexec`

给服务器提交一条命令并且等待结果。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

返回一个 `PGresult` 指针或者也可能是一个空指针。 通常返回一个非空指针，除非耗尽内存或发生了像不能把命令发送到服务器这样的严重错误。 应该调用 `PQresultStatus` 函数来检查任何错误的返回值 （包括空指针的值，在这种情况下它将返回 `PGRES_FATAL_ERROR` ）。 使用 `PQerrorMessage` 获取有关错误的更多信息。

命令字符串可以包括多个SQL命令（用分号分隔）。在一个 `PQexec` 调用中发送的多个查询是在一个事务里处理的，除非在查询字符串里有明确的 `BEGIN` / `COMMIT` 命令把整个字符串分隔成多个事务。 请注意，返回的 `PGresult` 结构只描述字符串里执行的最后一条命令的结果。 如果有一个命令失败，那么字符串处理的过程就会停止，并且返回的 `PGresult` 会描述错误条件。

`PQexecParams`

向服务器提交一条命令并且等待结果，还有独立于SQL命令文本传递参数的能力。

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` 类似 `PQexec` ，但是提供了额外的功能： 参数值可以独立于命令字符串进行声明，并且可以要求查询结果的格式是文本或二进制的。 `PQexecParams` 只是在协议3.0及以后的版本中支持；在使用协议2.0的时候会失败。

函数的参数是：

`conn`

连接对象通过它发送命令。

`command`

要执行的SQL命令字符串。如果使用参数，它们在命令字符串中被叫做 `$1` 、 `$2` 等等。

`nParams`

提供的参数数目；它是 `paramTypes[]` 、 `paramValues[]` 、 `paramLengths[]` 和 `paramFormats[]` 数组的长度。 （当 `nParams` 是0时，数组指针可以是 `NULL` 。）

`paramTypes[]`

通过OID，将声明数据类型指定到参数标记。如果 `paramTypes` 是 `NULL` ，或数组中任何的特定参数是0，服务器为参数标记推断数据类型， 采用的方式与一个未定义类型的文本字符串相同。

`paramValues[]`

声明参数的实际值。在这个数组中的一个空指针表示相应的参数是空； 否则指针指向一个以零结尾的文本字符串（文本格式）或者服务器希望的格式的二进制数据 （二进制格式）。

`paramLengths[]`

为二进制格式的参数声明实际数据长度。该设置忽略空参数或文本格式的参数。 如果没有二进制参数，那么数组指针可以为空。

`paramFormats[]`

声明参数为文本（为相应参数在数组条目中放置一个0）还是二进制格式 （为相应参数在数组条目中放置一个1）。如果数组指针是空，那么所有参数被看做是文本字符串。

以二进制格式传递的值需要能够被后台识别的内部表示。例如，整数必须以网络字节顺序来传递。 传递 `numeric` 值需要服务器存储格式的识别，如在 `src/backend/utils/adt/numeric.c::numeric_send()` 和 `src/backend/utils/adt/numeric.c::numeric_recv()` 中那样。

`resultFormat`

声明0用于以文本格式获得结果，或1用于以二进制格式获得结果。 （目前没有规定以不同的格式来获取不同的结果列，即使底层协议中可能实现。）

`PQexecParams` 相比 `PQexec` 的主要优势是参数值可以从命令字符串中分离出来， 因此避免了繁琐和容易出错的引用和逃逸的需要。

和 `PQexec` 不同的是， `PQexecParams` 在一个给出的字符串里最多允许一个SQL命令。 （里面可以有分号，但是不得超过一个非空的命令。）这是下层协议的一个限制， 但是也有些好处，比如作为对SQL注入攻击的额外防御。

> **Tip:** 通过OID声明参数类型是非常繁琐的，尤其是你不希望在你的程序里写死特定的OID值的时候。 不过，你可以避免这么做，即使在服务器自己无法判断参数类型， 或者是选择了一种与你预期不同的参数类型的时候也一样。在SQL命令文本里， 给参数符号附加一个明确的类型转换，显示你准备发送的数据类型。比如：
>
> ```
> SELECT * FROM mytable WHERE x = $1::bigint;
> ```
>
> 这样强制参数 `$1` 当作 `bigint` 看待，即使缺省情况下它会被赋予和 `x` 一样的类型。在以二进制格式发送参数值的时候， 我们强烈建议通过这种方法或者是声明数字类型OID的方法强制类型判断， 因为二进制格式比文本格式少一些冗余，因此服务器就会少一些机会捕捉类型的错误匹配。

`PQprepare`

用给定的参数提交请求，创建一个预备语句，然后等待结束。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

`PQprepare` 创建一个为后面 `PQexecPrepared` 执行用的预备语句。 这个特性允许那些重复使用的语句只分析和规划一次，而不是每次执行都分析规划。 只是在协议3.0和以后的连接里支持 `PQprepare` ；在使用2.0协议的时候，它会失败。

这个函数从 `query` 字串里创建一个叫 `stmtName` 的预备语句， `query` 必须只包含一个 SQL 命令。 `stmtName` 可以是 `""` ， 这样就创建一个无名的语句，这种情况下，任何前面存在的无名语句都会自动被代替； 否则，如果语句名已经在当前会话里定义，那就是一个错误。如果使用了参数， 那么在查询里它们引用成 `$1` ， `$2` 等等。 `nParams` 是参数的个数，参数的类型在数组 `paramTypes[]` 里事先声明好了。 （如果 `nParams` 是零，那么这个数组指针可以是 `NULL` 。） `paramTypes[]` 用 OID 的方式声明与参数符号关联的数据类型。 如果 `paramTypes` 为 `NULL` ，或者数组中某个特定元素是零， 那么服务器将用处理无类型文本同样的方法给这个参数符号赋予数据类型。还有， 查询可以使用比 `nParams` 数值更大的参数符号编号； 也为这些符号推断数据类型。 （参阅 `PQdescribePrepared` 作为一个找出推断的什么类型的手段。）

和 `PQexec` 相似，结果通常是一个 `PGresult` 对象， 其内容表明服务器端是成功还是失败。空的结果表示内存耗尽或者完全不能发送命令。 使用 `PQerrorMessage` 获取有关这类错误的更多信息。

用于 `PQexecPrepared` 的预备语句也可以通过执行SQL PREPARE语句来创建。 还有，尽管没有libpq函数可以删除一个预备语句， SQL DEALLOCATE语句却可以删除。

`PQexecPrepared`

发送一个请求，执行一个带有给出参数的预备语句，并且等待结果。

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecPrepared` 和 `PQexecParams` 类似， 但是要执行的命令是通过命名一个前面准备好的语句声明的，而不是给出一个查询字串。 这个特性允许那些要重复使用的命令只进行一次分析和规划，而不是每次执行都来一遍。 这个语句必须在当前会话的前面已经准备好。 `PQexecPrepared` 只在协议 3.0 和以后的版本里支持；在使用 2.0 版本的协议的时候，它们会失败。

参数和 `PQexecParams` 一样，只是给出的是一个预备语句的名字，而不是一个查询字串， 并且没有 `paramTypes[]` 参数（没必要，因为预备语句的参数类型是在创建的时候确定的）。

`PQdescribePrepared`

提交请求以获取有关指定的预备语句的信息，并等待完成。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

`PQdescribePrepared` 允许应用程序获取有关先前准备的语句的信息。 `PQdescribePrepared` 只在协议 3.0 和以后的版本里支持； 在使用 2.0 版本的协议的时候，它们会失败。

`stmtName` 可以是 `""` 或 `NULL` 以指向未命名声明， 要么必须与现有的预备语句同名。成功时，会返回一个带有 `PGRES_COMMAND_OK` 的 `PGresult` 。可以在这个 `PGresult` 中使用 `PQnparams` 和 `PQparamtype` 函数以获得预备语句的参数信息，同时 `PQnfields` ， `PQfname` ， `PQftype` 等函数提供声明的结果列（如果有）的信息。

`PQdescribePortal`

提交请求以获取有关指定的端口的信息，并等待完成。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

`PQdescribePortal` 允许应用程序获得关于之前创建的端口的信息。 （libpq不提供与端口的直接连接，但可以使用这个函数来检查 `DECLARE CURSOR` 命令创建的游标的属性）。 `PQdescribePortal` 只支持3.0及其之后的连接协议；当使用协议2.0时会失败。

`portalName` 可以是 `""` 或 `NULL` 以指向未命名声明， 要么必须与现有的预备语句同名。成功时，会返回一个带有 `PGRES_COMMAND_OK` 的 `PGresult` 。可以在 `PGresult` 中使用 `PQnfields` ， `PQfname` ， `PQftype` 等函数获取端口的结果列（如果有）的信息。

`PGresult` 结构封装了服务器返回的结果。libpq应该小心维护 `PGresult` 的抽象。使用下面的访问函数获取 `PGresult` 的内容。避免直接引用 `PGresult` 里面的字段，因为它们在未来版本里可能会被修改。

`PQresultStatus`

返回命令的结果状态。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

`PQresultStatus` 可以返回下面数值之一：

`PGRES_EMPTY_QUERY`

发送给服务器的字串是空的。

`PGRES_COMMAND_OK`

成功完成一个不返回数据的命令。

`PGRES_TUPLES_OK`

成功执行一个返回数据的查询（比如 `SELECT` 或者 `SHOW` ）。

`PGRES_COPY_OUT`

（从服务器）Copy Out （拷贝出）数据传输开始。

`PGRES_COPY_IN`

Copy In（拷贝入）（到服务器）数据传输开始。

`PGRES_BAD_RESPONSE`

服务器的响应无法理解。

`PGRES_NONFATAL_ERROR`

发生了一个非致命错误（通知或者警告）。

`PGRES_FATAL_ERROR`

发生了一个致命错误。

`PGRES_COPY_BOTH`

拷贝入/出（到和从服务器）数据传输开始。这个特性当前只用于流复制， 所以这个状态不会在普通应用中发生。

`PGRES_SINGLE_TUPLE`

`PGresult` 包含一个来自当前命令的结果元组。 这个状态只在查询选择了单行模式时发生（参阅Section 31.5）。

如果结果状态是 `PGRES_TUPLES_OK` 或 `PGRES_SINGLE_TUPLE` ， 那么可以用下面的函数从查询的返回中抽取元组信息。注意一个碰巧检索了零条元组的 `SELECT` 仍然显示 `PGRES_TUPLES_OK` 。 `PGRES_COMMAND_OK` 用于不返回元组的命令（没有 `RETURNING` 子句的 `INSERT` ， `UPDATE` 等）。 返回 `PGRES_EMPTY_QUERY` 的响应通常意味着暴露了客户端软件里面的臭虫。

状态为 `PGRES_NONFATAL_ERROR` 的结果永远不会直接由 `PQexec` 或者其它查询执行函数返回；这类的结果会被传递给通知处理器 （参阅Section 31.12）。

`PQresStatus`

把 `PQresultStatus` 返回的枚举类型转换成一个描述状态码的字符串常量。 调用者不应该释放结果。

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`

返回与查询关联的错误信息，或在没有错误时返回一个空字符串。

```
char *PQresultErrorMessage(const PGresult *res);
```

如果有错误，那么返回的字串将包括一个结尾的新行。调用者不应该直接释放结果。 在相关的 `PGresult` 句柄传递给 `PQclear` 之后，它会自动释放。

紧跟在一个 `PQexec` 或 `PQgetResult` 调用后面， `PQerrorMessage` （对连接）将返回与 `PQresultErrorMessage` (对结果)一样的字符串。不过，一个 `PGresult` 将保有其错误信息直到被删除， 而连接的错误信息将在后续的操作完成时被改变。当你想知道与某个 `PGresult` 相关联的状态时用 `PQresultErrorMessage` ； 当你想知道与连接的最近一个操作相关联的状态时用 `PQerrorMessage` 。

`PQresultErrorField`

返回一个独立的错误报告字段。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode` 是一个错误字段标识符；参阅下面列出的符号。 如果 `PGresult` 不是错误或者警告结果或者不包括指定的字段， 那么返回 `NULL` 。字段值通常将不包括结尾的新行。调用者不应该直接释放结果。 在相关联的 `PGresult` 句柄传递给 `PQclear` 之后，它将被自动释放。

下列代码是可用的：

`PG_DIAG_SEVERITY`

严重程度，这个字段的内容是`ERROR`，`FATAL`或者`PANIC`（在错误信息里），或者`WARNING`，`NOTICE`，`DEBUG`，`INFO`或`LOG`（在注意信息里），或者是这些东西的一个本地化翻译。总是出现。

`PG_DIAG_SQLSTATE`

这个错误的SQLSTATE代码。SQLSTATE代码表示所发生的错误的类型；可以由前端应用用于对特定的数据库错误执行特定的操作（比如错误处理）。可能的SQLSTATE代码的列表，请查看Appendix A。这个字段是不能区域化的，并且总是出现。

`PG_DIAG_MESSAGE_PRIMARY`

主要的人类可读错误的信息（通常一行）。总是出现。

`PG_DIAG_MESSAGE_DETAIL`

细节：一个可选的从属错误信息，里面有更多有关该问题的细节。可能有多行。

`PG_DIAG_MESSAGE_HINT`

提示：一个可选的有关如何处理该问题的建议。它和细节的区别是它提供了建议（可能不太合适）而不光是事实。可能有好几行。

`PG_DIAG_STATEMENT_POSITION`

一个包含十进制整数的字串，表明错误游标的位置，作为一个索引指向最初的语句字符串。第一个字符的索引是 1，并且这个位置是用字符计，而不是用字节计。

`PG_DIAG_INTERNAL_POSITION`

这个和`PG_DIAG_STATEMENT_POSITION`字段定义是一样的，区别是它在游标位置指向内部生成的命令时使用，而不是客户端提交的命令。如果出现了这个字段，那么`PG_DIAG_INTERNAL_QUERY`字段也总是出现。

`PG_DIAG_INTERNAL_QUERY`

一个失败的内部生成的命令的文本。比如，这个可能是一个 PL/pgSQL 函数发出的 SQL 查询。

`PG_DIAG_CONTEXT`

一个指示器，表明错误发生的环境。目前这个包括活跃的过程语言函数和内部生成的查询的调用堆栈跟踪。跟踪是每行一条，最近的在上面。

`PG_DIAG_SCHEMA_NAME`

如果错误与特定的数据库对象相关，那么是包含该对象的模式名（如果有）。

`PG_DIAG_TABLE_NAME`

如果错误与特定的表相关，那么是该表的名字。（参考模式名字段获取表的模式的名字。）

`PG_DIAG_COLUMN_NAME`

如果错误与特定的表字段相关，那么是该字段的名字。（参考模式和表名字段识别该表。）

`PG_DIAG_DATATYPE_NAME`

如果错误与特定的数据类型相关，那么是该数据类型的名字。（参考模式名字段获取数据类型的模式的名字。）

`PG_DIAG_CONSTRAINT_NAME`

如果错误与特定的约束相关，那么是该约束的名字。参考上面列出的字段获取相关的表或域。（为了这个目的，索引被看做是约束，即使它们是用约束语法创建的。）

`PG_DIAG_SOURCE_FILE`

报告错误的源代码所在的文件名。

`PG_DIAG_SOURCE_LINE`

报告错误的源代码所在的行号。

`PG_DIAG_SOURCE_FUNCTION`

报告错误的源代码函数的名字。

> **Note:** 只为有限的错误类型提供模式名、表名、字段名、数据类型名和约束名字段；参阅Appendix A。不要假设这些字段的出现会保证其他字段的出现。核心错误来源观察以上提到的相互关系，但是用户定义的函数可以以其他方式使用这些字段。同样的，不要假设这些字段表示当前数据库中的同时期对象。

按照自身的要求格式化显示信息是客户端的责任；特别是根据需要对长行进行折行。在错误信息字段里出现的新行字符应该当作分段符号，而不是换行。

libpq生成的错误将会有严重性和主信息，但是通常没有其它字段。3.0 协议之前返回的错误将包含严重性和主信息，有时候还有详细信息，但是没有其它字段。

请注意这些错误字段只能从 `PGresult` 对象里获得，而不是 `PGconn` 对象；没有 `PQerrorField` 函数。

`PQclear`

释放与 `PGresult` 相关联的存储空间。任何不再需要的查询结果都应该用 `PQclear` 释放掉。

```
void PQclear(PGresult *res);
```

只要你需要，你可以保留 `PGresult` 对象任意长的时间；当你提交新的查询时它并不消失，甚至你断开连接后也是这样。要删除它，你必须调用 `PQclear` 。不这么做将导致你应用中的内存泄漏。

31.3.2. 检索查询结果信息

这些函数用于从一个代表着成功查询结果（也就是说，状态为 `PGRES_TUPLES_OK` 或 `PGRES_SINGLE_TUPLE` 的查询）的 `PGresult` 对象中抽取信息。 它们也可以用于从一个成功描述操作中抽取信息： 一个描述的结果和实际查询的执行将要提供的结果有所有相同的字段信息，但是它有零行。 对于其它状态值的对象，他们的行为会好像他们有零行和零列一样。

`PQntuples`

返回查询结果里的行（元组）个数。因为它返回一个整数的结果，在32位操作系统上大型结果集可能溢出返回值。

```
int PQntuples(const PGresult *res);
```

`PQnfields`

返回查询结果里数据行的列（字段）的个数。

```
int PQnfields(const PGresult *res);
```

`PQfname`

返回与给出的字段编号相关联的字段名。字段编号从 0 开始。调用者不应该直接释放结果。在相关联的 `PGresult` 句柄传递给 `PQclear` 之后，结果会被自动释放。

```
char *PQfname(const PGresult *res,
              int column_number);
```

如果字段编号超出范围，那么返回 `NULL` 。

`PQfnumber`

返回与给出的字段名相关的字段编号。

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

如果给出的名字不匹配任何字段，返回-1。

给出的名字是当作 SQL 命令里的一个标识符看待的，也就是说，如果没有加双引号， 那么会转换为小写。比如，如果我们有一个从 SQL 命令里生成的查询结果：

```
SELECT 1 AS FOO, 2 AS "BAR";
```

那么我们会有下面的结果：

```
PQfname(res, 0)              _foo_
PQfname(res, 1)              _BAR_
PQfnumber(res, "FOO")        _0_
PQfnumber(res, "foo")        _0_
PQfnumber(res, "BAR")        _-1_
PQfnumber(res, "\"BAR\"")    _1_
```

`PQftable`

返回我们抓取的字段所在的表的 OID。字段编号从 0 开始。

```
Oid PQftable(const PGresult *res,
             int column_number);
```

如果字段编号超出了范围，或者声明的字段不是一个指向某个表的字段的简单引用， 或者使用了 3.0 版本之前的协议，那么就会返回 `InvalidOid` 。 你可以查询系统表 `pg_class` 来判断究竟引用了哪个表。

在你包含libpq头文件的时候，就会定义类型 `Oid` 和常量 `InvalidOid` 。他们都是相同的整数类型。

`PQftablecol`

返回组成声明的查询结果字段的字段号（在它的表内部）。查询结果字段编号从 0 开始， 但是表字段编号不会是 0。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

如果字段编号超出范围，或者声明的字段并不是一个表字段的简单引用， 或者使用的是 3.0 之前的协议，那么返回零。

`PQfformat`

返回说明给出字段的格式的格式代码。字段编号从 0 开始。

```
int PQfformat(const PGresult *res,
              int column_number);
```

格式码为 0 表示文本数据，而格式码是一表示二进制数据。（其它编码保留给将来定义。）

`PQftype`

返回与给定字段编号关联的数据类型。返回的整数是一个该类型的内部 OID 号。字段编号从0开始。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

你可以查询系统表`pg_type`以获取各种数据类型的名称和属性。 内建的数据类型的OID在源码树的 `src/include/catalog/pg_type.h` 文件里定义。

`PQfmod`

返回与给定字段编号相关联的字段的类型修饰符。字段编号从 0 开始。

```
int PQfmod(const PGresult *res,
           int column_number);
```

类型修饰符的值是类型相关的；他们通常包括精度或者尺寸限制。 数值 -1 用于表示"没有可用信息"。大多数数据类型不用修饰词，这种情况下该值总是-1。

`PQfsize`

返回与给定字段编号关联的字段以字节计的大小。字段编号从0 开始。

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize`返回在数据库行里面给该数据字段分配的空间， 换句话说就是该数据类型在服务器的内部表现形式的大小（尺寸）。（因此， 这个对客户端没有什么用。） 负值表示该数据类型是可变长度。

`PQbinaryTuples`

如果 `PGresult` 包含二进制数据时返回 1，如果包含文本数据返回 0。

```
int PQbinaryTuples(const PGresult *res);
```

这个函数已经废弃了（除了还用于与`COPY`连接之外）， 因为我们可能在一个`PGresult`的某些字段里包含文本数据， 而另外一些字段包含二进制数据。更好的是使用`PQfformat`。 `PQbinaryTuples`只有在结果中的所有字段都是二进制（格式 1）的时候才返回 1。

`PQgetvalue`

返回一个`PGresult`里面一行的单独的一个字段的值。 行和字段编号从 0 开始。调用者不应该直接释放结果。在把`PGresult` 句柄传递给`PQclear`之后，结果会被自动释放。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

对于文本格式的数据， `PQgetvalue`返回的值是一个表示字段值的空（NULL） 结尾的字符串。对于二进制格式，返回的值就是由该数据类型的`typsend` 和 `typreceive`决定的二进制表现形式。（在这种情况下， 数值实际上也跟着一个字节零，但是通常这个字节没什么用处，因为数值本身很可能包含内嵌的空。）

如果字段值是空，则返回一个空字串。参阅 `PQgetisnull` 来区别空值和空字串值。

`PQgetvalue` 返回的指针指向一个本身是 `PGresult` 结构的一部分的存储区域。我们不能更改它，并且如果我们要在 `PGresult` 结构的生存期后还要使用它的话，我们必须明确地把该数值拷贝到其他存储器中。

`PQgetisnull`

测试一个字段是否为空（NULL）。行和字段编号从 0 开始。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

如果该域包含 NULL，函数返回 1，如果包含非空（non-null ）值，返回 0。 （注意，对一个 NULL 字段， `PQgetvalue` 将返回一个空字符串，不是一个空指针。）

`PQgetlength`

返回以字节计的字段的长度。行和字段编号从 0 开始。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

这是特定数值的实际数据长度，也就是说， `PQgetvalue` 指向的对象的大小。 对于文本数据格式，它和 `strlen()` 相同。对于二进制格式，这是基本信息。 请注意我们不应该依靠 `PQfsize` 获取实际数据长度。

`PQnparams`

返回一个预备语句中的参数的数目。

```
int PQnparams(const PGresult *res);
```

只有在检查 `PQdescribePrepared` 的结果时，这个函数是有用的。 对于其他类型的查询将返回零。

`PQparamtype`

返回指示语句中的参数的数据类型。参数编号从0开始。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

只有在检查 `PQdescribePrepared` 的结果时，这个函数是有用的。 对于其他类型的查询将返回零。

`PQprint`

向指定的输出流打印所有的行和（可选的）字段名称。

```
void PQprint(FILE *fout,      /* 输出流 */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool  header;      /* 打印输出字段头和行计数 */
    pqbool  align;       /* 填充对齐字段 */
    pqbool  standard;    /* 旧的格式 */
    pqbool  html3;       /* 输出HTML表 */
    pqbool  expanded;    /* 扩展表 */
    pqbool  pager;       /* 必要时在输出中使用分页器 */
    char    *fieldSep;   /* 字段分隔符 */
    char    *tableOpt;   /* HTML表格元素的属性 */
    char    *caption;    /* HTML表标题 */
    char    **fieldName; /* 替换字段名组成的空结尾的数组 */
} PQprintOpt;
```

这个函数以前被psql用于打印查询结果，但是现在已经不用这个函数了。 请注意它假设所有的数据都是文本格式。

31.3.3. 检索其它命令的结果信息

这些函数用于从 `PGresult` 对象里检索其他信息。

`PQcmdStatus`

返回产生 `PGresult` 的 SQL 命令的命令状态标签。

```
char *PQcmdStatus(PGresult *res);
```

通常这只是命令的名字，但是它可能包括额外的数据，比如处理过的行数。调用者不应该直接释放结果。 结果会在把 `PGresult` 句柄传递给 `PQclear` 的时候释放。

`PQcmdTuples`

返回被 SQL 命令影响的行的数量。

```
char *PQcmdTuples(PGresult *res);
```

这个函数返回一个字符串，包含 `PGresult` 产生的SQL语句影响的行数。 这个函数只能用于下列的执行： `SELECT` ， `CREATE TABLE AS` ， `INSERT` ， `UPDATE` ， `DELETE` ， `MOVE` ， `FETCH` ，或者 `COPY` 语句，或者是一个包含 `INSERT` ， `UPDATE` 或 `DELETE` 语句的预备查询的 `EXECUTE` 。如果生成这个 `PGresult` 的命令是其他的东西， 那么 `PQcmdTuples` 返回一个空字串。调用者不应该直接释放返回的数值。 在相关联的 `PGresult` 被传递给 `PQclear` 之后，它会被自动释放。

`PQoidValue`

返回插入的行的OID， 如果SQL命令是 `INSERT` ，插入了正好一行到有OID的表格， 或者是一个包含合适 `INSERT` 语句的预备查询 `EXECUTE` 的时候。 否则，函数返回 `InvalidOid` 。如果受 `INSERT` 影响的表不包含 OID，也返回 `InvalidOid` 。

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

为了支持 `PQoidValue` ，这个函数已经废弃了，并且不是线程安全的。 它返回插入行的带有OID的字符串，而 `PQoidValue` 返回OID值。

```
char *PQoidStatus(const PGresult *res);
```

31.3.4. 逃逸包含在SQL命令中的字符串

`PQescapeLiteral`

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

`PQescapeLiteral` 为在 SQL 命令中使用字串而对之进行逃逸处理。 在我们向 SQL 命令里把数据值当作文本常量插入的时候很有用。有些字符 （比如单引号和反斜杠）必须被逃逸，以避免他们被 SQL 分析器作为特殊字符解析。 `PQescapeLiteral` 执行这个操作。

`PQescapeLiteral` 返回一个内存中分配有 `malloc()` 的 `str` 参数的逃逸版本。当结果不再需要时，需要通过 `PQfreemem()` 来释放这块内存。不需要一个0字节结束，并且不应以 `length` 计数。 （如果在处理 `length` 字节之前出现0字节的结束， `PQescapeLiteral` 在此处结束；这个行为有点像 `strncpy` ）。返回的字符串中所有特殊字符都替换掉了， 因此可以很好的被PostgreSQL字符串文本解析器处理，同样， 允许增加一个0字节结尾。必须在PostgreSQL字符串文本两边的单引号包含在结果字符串中。

一旦错误， `PQescapeLiteral` 返回 `NULL` 并在 `conn` 对象中存储合适的信息。

> **Tip:** 处理从不可信来源收到的字符串时必须进行合适的逃逸，否则存在一定的安全风险： 容易受到"SQL 注入"攻击，数据库中会被写入未知的SQL命令。

需要注意的是，当一个数据以 `PQexecParams` 或它的兄弟格式， 作为一个单独的参数传递时，做逃逸是不必要，也是不正确的。

`PQescapeIdentifier`

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

`PQescapeIdentifier` 逃逸一个字符串作为一个SQL标识符使用， 如一个表，列，或函数名。当一个用户自定义标识符需要包含特殊的字符， 否则将不能被SQL解析器解析为标识符的一部分时，或者当标识符需要包含大写字母， 且这种情况必须保留时，这样做是很有用的。

`PQescapeIdentifier` 返回 `str` 参数逃逸为一个内存中分配有 `malloc()` 的SQL标识符的版本。当结果不再需要时，这块内存必须使用 `PQfreemem()` 来释放。不需要一个0字节结束，并且不应以 `length` 计数。 (如果在处理 `length` 字节之前出现0字节的结束， `PQescapeIdentifier` 在此处结束；这个行为比较像 `strncpy` ）。返回的字符串中所有特殊字符都替换掉了， 因此可以很好的作为SQL标识符被处理。也可以添加一个结尾的0字节。返回字符串也是被双引号环绕。

出错时， `PQescapeIdentifier` 返回 `NULL` ， 并且在 `conn` 对象中存贮合适的信息。

> **Tip:** 由于带有字符串常量，为阻止SQL注入攻击，当从一个不可信任资源获得时，SQL标识符必须逃逸。

`PQescapeStringConn`

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

`PQescapeStringConn` 逃逸字符串常量，比较像 `PQescapeLiteral` 。 不同于 `PQescapeLiteral` ，请求应该提供一个适当大小的缓冲区。更重要的是， `PQescapeStringConn` 不会生成一个必须在PostgreSQL 字符串常量两端的单引号；SQL命令中应该提供，这样结果中会被插入。 `from` 参数指向字符串的第一个字符（用以逃逸）， `length` 参数指出了在这个字符串中的字节数。 不需要一个0字节结束，并且不应以 `length` 计数。 (如果在处理 `length` 字节之前出现0字节的结束， `PQescapeStringConn` 在此处结束；这个行为比较像 `strncpy` ）。 `to` 应该指向一个包含至少多于两倍 `length` 大小的缓冲区，要么就不会定义该行为。如果 `to` 和 `from` 字符串交叠，那么也不会定义该行为。

`error` 参数非 `NULL` ， 那么在成功的时候 `*error` 会被设置为零， 失败的时候设置为非0。目前唯一可能的错误条件涉及在源字符串中无效的多字节编码。 输出字符串同样产生错误，但服务器可以视其为异常以拒绝。一旦发生错误， 一条合适的信息会存储在 `conn` 对象中，无论 `error` 是否为 `NULL` 。

`PQescapeStringConn` 返回写到 `to` 的字节数，不包含0字节终止。

`PQescapeString`

`PQescapeString` 是一个老的，已经被 `PQescapeStringConn` 弃用了的版本。

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

与 `PQescapeStringConn` 唯一的不同是， `PQescapeString` 不使用 `PGconn` 或 `error` 参数。因此，不能够根据连接属性（如字符编码）来调整其行为，因此可能会给出错误的结果，同样，不会报告错误条件。

`PQescapeString` 可以在客户端编程（一次只有一个PostgreSQL连接） 中安全的使用。在这种情况下，它可以找到"在屏幕背后"想要知道的。在其他情况下， 这是一个安全隐患，使用 `PQescapeStringConn` 时应该避免。

`PQescapeByteaConn`

逃逸那些在 SQL 命令中使用的用 `bytea` 表示的二进制数据。 和 `PQescapeStringConn` 一样，这个函数只有在直接向 SQL 字串插入数据的时候使用。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

在SQL语句中用做 `bytea` 字串文本的一部分的时候， 有些字节值必需逃逸。 `PQescapeByteaConn` 逃逸字节使用十六进制编码或反斜杠逃逸。 参阅Section 8.4获取更多信息。

`from` 参数指向需要逃逸的字串的第一个字节， `from_length` 参数反映在这个二进制字串（结尾的字节零既不必要也不计算在内）里字节的个数。 `to_length` 参数指向一个变量，它保存逃逸后字符串长度的结果。 结果字串长度包括结果结尾的零字节。

`PQescapeByteaConn` 在内存中返回一个 `from` 参数的二进制字串的逃逸后的版本，这片内存是用 `malloc()` 分配的 在不再需要结果的时候，必须用 `PQfreemem()` 释放内存。 返回的字串已经把所有特殊的字符替换掉了，这样他们就可以由PostgreSQL 的字串文本分析器以及 `bytea` 的输入函数正确地处理。同时还追加了一个结尾的字节零。 那些必需包围在PostgreSQL字串文本周围的单引号并非结果字串的一部分。

当出错时，返回一个空指针，一个合适的错误信息会被储存在 `conn` 对象中， 当前唯一可能的错误是结果字符串的内存不足。

`PQescapeBytea`

`PQescapeBytea` 是 `PQescapeByteaConn` 的一个旧的，过时的版本。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

与 `PQescapeByteaConn` 唯一的不同之处在于， `PQescapeByteaConn` 不使用 `PGconn` 参数，因此， `PQescapeBytea` 可以在客户端编程 （一次只有一个PostgreSQL连接）中安全的使用。在这种情况下， 它可以找到"在屏幕背后"想要知道的。如果在编程中使用多个数据库连接 （在这种情况下使用 `PQescapeByteaConn` ），那么可能会给出错误结果。

`PQunescapeBytea`

把一个二进制数据的字符串表现形式转换成二进制数据— `PQescapeBytea` 的反作用。在以文本格式抽取 `bytea` 数据的时候是必须的，但是在以二进制格式抽取的时候是不必要的。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

`from` 参数指向一个字符串，比如应用到 `bytea` 字段时， `PQgetvalue` 返回的。 `PQunescapeBytea` 把它的字串表现形式转换成二进制形式，它返回一个用 `malloc()` 分配的指向该缓冲区的指针，或者是出错时返回 `NULL` ，缓冲区的尺寸放在 `to_length` 里。在不再需要这个结果之后， 这片内存必须用 `PQfreemem` 释放。

这个转换不正好是 `PQescapeBytea` 逆转换，因为，当从 `PQgetvalue` 接收时，字符串不希望被"逃逸"。尤其是，这意味着， 不需要考虑字符串引用，并且不需要 `PGconn` 参数。

31.4. 异步命令处理

`PQexec` 函数对普通的同步应用里提交命令已经是足够用的了。 但是它却有几个缺陷，而这些缺陷可能对某些用户很重要：

- `PQexec` 等待命令结束。而应用可能还有其它的工作要做（比如维护用户界面等），这个时候它可不想阻塞在这里等待响应。
- 因为客户端应用在等待结果的时候是处于挂起状态的，所以应用很难判断它是否该尝试结束正在进行的命令。（这个事情可以在一个信号处理器中做，但是没别的方法。）
- `PQexec` 只能返回一个 `PGresult` 结构。 如果提交的命令字符串包含多个SQL命令，除了最后一个 `PGresult` 以外都会被 `PQexec` 丢弃。
- `PQexec` 总是收集命令的整个结果，将其缓存在一个 `PGresult` 中。 虽然这为应用简化了错误处理逻辑，但是对于包含多行的结果是不切实际的。

不想受到这些限制的应用可以改用下面的函数，这些函数也是构造 `PQexec` 的函数：`PQsendQuery` 和 `PQgetResult` 。 也有 `PQsendQueryParams` ， `PQsendPrepare` ， `PQsendQueryPrepared` ， `PQsendDescribePrepared` 和 `PQsendDescribePortal` ，他们可以和 `PQgetResult` 一起使用，分别用于复制 `PQexecParams` ， `PQprepare` ， `PQexecPrepared` ， `PQdescribePrepared` 和 `PQdescribePortal` 的功能。

`PQsendQuery`

向服务器提交一个命令而不等待结果。如果查询成功发送则返回 1，否则返回 0。 （此时，可以用 `PQerrorMessage` 获取关于失败的信息）。

```
int PQsendQuery(PGconn *conn, const char *command);
```

在成功调用 `PQsendQuery` 后，调用 `PQgetResult` 一次或者多次获取结果。在 `PQgetResult` 返回 NULL 指针， 表明命令完成之前，我们不能再调用 `PQsendQuery` （在同一次连接里）。

`PQsendQueryParams`

给服务器提交一个命令和分隔的参数，而不等待结果。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

这个等效于 `PQsendQuery`，只是查询参数可以和查询字串分开声明。 函数的参数处理和 `PQexecParams` 一样。 和 `PQexecParams` 类似，它不能在 2.0 版本的协议连接上工作， 并且它只允许在查询字串里出现一条命令。

`PQsendPrepare`

发送一个请求，创建一个给定参数的预备语句，而不等待结束。

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

这是 `PQprepare` 的异步版本：如果它能发送这个请求，则返回 1，如果不能，则返回 0。 在成功调用之后，调用 `PQgetResult` 判断服务器是否成功创建了预备语句。 这个函数的参数的处理和 `PQprepare` 一样。类似 `PQprepare` ， 它不能在 2.0 版本协议的连接上运转。

`PQsendQueryPrepared`

发送一个请求执行带有给出参数的预备语句，不等待结果。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

这个函数类似 `PQsendQueryParams` ， 但是要执行的命令是通过给一个前面准备好的语句命名来声明的，而不是给出一个查询字串。 函数的参数处理和 `PQexecPrepared` 一样。 类似 `PQexecPrepared` ，它也不能在 2.0 版本的协议连接上工作。

`PQsendDescribePrepared`

提交一个请求，获取关于指定的预备语句的信息，不等待结果。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

这是 `PQdescribePrepared` 的一个异步版本：如果它能发送这个请求，则返回 1， 如果不能，则返回 0。在成功调用之后，调用 `PQgetResult` 获取结果。 这个函数的参数的处理和 `PQdescribePrepared` 一样。 类似 `PQdescribePrepared` ，它不能在 2.0 版本协议的连接上运转。

`PQsendDescribePortal`

发出请求，以获得关于指定端口的信息，不需要等待完成。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

这是一个 `PQdescribePortal` 的异步版本：如果它能发送这个请求，那么返回1， 否则返回0。成功调用之后，通过 `PQgetResult` 获得结果。 函数参数处理与 `PQdescribePortal` 相同。 类似于 `PQdescribePortal` ， 不能在2.0的协议连接上工作。

`PQgetResult`

等待从前面 `PQsendQuery` ， `PQsendQueryParams` ， `PQsendPrepare` ， `PQsendQueryPrepared` ， `PQsendDescribePrepared` 或者 `PQsendDescribePortal` 调用返回的下一个结果，然后返回之。当命令结束并且没有更多结果后返回 NULL。

```
PGresult *PQgetResult(PGconn *conn);
```

必须重复的调用 `PQgetResult` ， 直到它返回空指针，表明该命令结束。 （如果在没有活跃的命令时调用， `PQgetResult` 将只是立即返回一个空指针。） 每个 `PQgetResult` 返回的非 NULL 结果都应该用前面描述的 `PGresult` 访问函数进行分析。不要忘了在结束分析后用 `PQclear` 释放每个结果对象。注意， `PQgetResult` 只是在有一个命令是活跃的而且必须返回数的据还没有被 `PQconsumeInput` 读取时阻塞。

> **Note:** 即使在 `PQresultStatus` 表明一个致命的错误时， 也应该调用 `PQgetResult` 直到它返回一个空指针， 以允许libpq完全的处理错误信息。

使用 `PQsendQuery` 和 `PQgetResult` 解决了 `PQexec` 的一个问题：如果一个命令字符串包含多个SQL命令，这些命令的结果可以独立的获得。（这样就允许一种简单的重叠处理模式， 顺便说一句：客户端可以处理一个命令的结果而服务器可以仍然在处理同一命令字符串后面的查询。）

另一个可以用 `PQsendQuery` 和 `PQgetResult` 获得的经常需要的特性是一次检索大型连续查询结果。这在 Section 31.5中讨论。

单独的，调用 `PQgetResult` 将仍然导致客户端阻塞， 直到服务器完成下一个SQL命令。可以通过适当的使用两个函数避免：

`PQconsumeInput`

如果存在服务器来的输入可用，则使用之。

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` 通常返回 1 表明"没有错误"， 而返回 0 表明有某种错误发生， （这个时候可以用 `PQerrorMessage` ）。 注意这个结果并不表明实际上是否收集了输入数据。在调用 `PQconsumeInput` 之后， 应用可以检查 `PQisBusy` 和/或 `PQnotifies` 看一眼它们的状态是否改变。

`PQconsumeInput` 可以在应用还没有做好处理结果或通知的情况下被调用。 这个函数将读取可用的数据并且在一个缓冲区里保存它，这样导致一个 `select()` 读准备好标识的生成。这样应用就可以使用 `PQconsumeInput` 立即清掉 `select()` 条件，然后在空闲的时候检查结果。

`PQisBusy`

在查询忙的时候返回 1 ，也就是说， `PQgetResult` 将阻塞住等待输入。 一个 0 的返回表明这时调用 `PQgetResult` 保证不阻塞。

```
int PQisBusy(PGconn *conn);
```

`PQisBusy` 本身将不会试图从服务器读取数据； 所以必须先调用 `PQconsumeInput` ， 否则将永远不会消除忙状态。

一个使用这些函数的典型的应用将有一个主循环使用 `select()` 或 `poll()` 等待所有它必须处理的条件。其中一个条件将会是服务器来的数据已准备好， 从 `select()` 的角度来看就是 `PQsocket` 标识的文件描述符上已经有可读取的数据。当主循环侦测到输入准备好， 它将调用 `PQconsumeInput` 读取输入。然后可以调用 `PQisBusy` ， 返回 false (0)后面可以跟着 `PQgetResult` 。同样它（用户应用） 可以调用 `PQnotifies` 检测 `NOTIFY` 信息 （参阅Section 31.8） 。

一个使用 `PQsendQuery` / `PQgetResult` 的客户端同样也可以试图取消一个正在被服务器处理的命令。参阅Section 31.6。 但是，不管 `PQcancel` 返回的值是多少， 应用都必须使用 `PQgetResult` 进行正常的读取结果的动作序列。 一次成功的取消只会导致命令比正常情况下快些结束。

通过使用上面描述的函数，我们可以避免在等待来自数据库服务器的输入时的阻塞。 不过，应用还是有可能阻塞在给服务器发送输出上。这种情况比较少见，但是也可能发生， 尤其是我们要发送非常长的 SQL 命令或者数据值的时候。 （不过， 最有可能的是在应用通过 `COPY IN` 发送数据的时候。） 为了避免这个可能性， 实现完全的非阻塞数据库操作，我们可以使用下列额外的函数。

`PQsetnonblocking`

把连接的状态设置为非阻塞。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

如果 `arg` 为 1，把连接状态设置为非阻塞， 如果 `arg` 为 0，把连接状态设置为阻塞。如果 OK 返回 0，如果错误返回 -1。

在非阻塞状态，调用 `PQsendQuery` ， `PQputline` ， `PQputnbytes` ， 和 `PQendcopy` 的时候不被阻塞， 而是在如果需要再次调用它们时将返回一个错误。

请注意 `PQexec` 不会在意任何非阻塞模式；如果调用了 `PQexec` ，那么它的行为总是阻塞的。

`PQisnonblocking`

返回数据库连接的阻塞状态。

```
int PQisnonblocking(const PGconn *conn);
```

如果连接设置为非阻塞状态，返回 1，如果是阻塞状态返回 0。

`PQflush`

试图把任何正在排队的数据冲刷到服务器，如果成功（或者发送队列为空）返回 0， 如果因某种原因失败返回 -1，或者是在无法把发送队列中的所有数据都发送出去，返回 1。 （这种情况只有在连接为不阻塞模式的时候才会出现）。

```
int PQflush(PGconn *conn);
```

在一个非阻塞的连接上发送任何命令或者数据之后，调用 `PQflush` 。 如果返回 1，就等待套接字写准备好然后再次调用；重复这个操作直到它返回 0。 一旦 `PQflush` 返回 0，则等待套接字为读准备好，准备好之后就像上面那样读取响应。

31.5. 逐行检索查询结果

通常，libpq收集SQL命令的全部结果并作为单个 `PGresult` 返回到应用中。这对于返回大量行的命令是不可能实现的。对于这种情况， 应用可以在单行模式中使用 `PQsendQuery` 和 `PQgetResult` 。 在这个模式中，结果行一次返回一行到应用中，就像从服务器中接收到它们一样。

要进入单行模式，在成功调用 `PQsendQuery` （或者一个兄弟函数） 之后立即调用 `PQsetSingleRowMode` 。这种模式选择只对当前执行的查询有效。 然后重复的调用 `PQgetResult` ， 直到它返回空，在Section 31.4 中记录。如果查询返回任意行， 它们作为独立的 `PGresult` 对象返回， 就像普通的查询结果，除了状态码是 `PGRES_SINGLE_TUPLE` 而不是 `PGRES_TUPLES_OK` 。在最后一行之后，或者一旦查询返回零行， 返回一个带有状态 `PGRES_TUPLES_OK` 的零行对象； 这是没有更多行的一个信号。 （但是，请注意，仍然需要继续调用 `PQgetResult` 直到它返回空。）所有这些 `PGresult` 对象将包含相同的描述数据 （字段名、 类型等）， 就像该查询的一个普通 `PGresult` 对象拥有的那样。 像往常一样，每个对象都应该使用 `PQclear` 释放。

`PQsetSingleRowMode`

为当前执行的查询选择单行模式。

```
int PQsetSingleRowMode(PGconn *conn);
```

这个函数只能在 `PQsendQuery` 或它的一个兄弟函数之后立即调用， 在任何连接上的其他操作，比如 `PQconsumeInput` 或 `PQgetResult` 之前。如果在正确的时间调用了， 该函数为当前查询激活单行模式并返回1。否则模式保持不变并返回0。在任何情况下， 该模式在当前查询完成之后恢复到正常。

Caution

处理一个查询时，服务器可能返回一些行然后遇到一个错误， 导致查询退出。通常，libpq丢弃任何这样的行并且只报告错误。但是在单行模式中， 这些行将早已返回到应用中。因此， 应用将看到一些 `PGRES_SINGLE_TUPLE` `PGresult` 对象跟随在 `PGRES_FATAL_ERROR` 对象后面。对于适当的事务行为， 如果查询最终失败了，那么应用必须设计为抛弃或撤销先前处理的行。

31.6. 取消正在处理的查询

一个客户端应用可以使用本节描述的函数，要求取消一个仍在被服务器处理的命令。

`PQgetCancel`

创建一个数据结构，这个数据结构包含通过特定数据库连接取消一个命令所需要的信息。

```
PGcancel *PQgetCancel(PGconn *conn);
```

给出一个 `PGconn` 连接对象， `PQgetCancel` 创建一个 `PGcancel` 对象。如果给出的 `conn` 是 `NULL` 或者是一个无效的连接，那么它将返回 `NULL` 。 `PGcancel` 对象是一个不透明的结构，不应该为应用所直接访问；我们只能把它传递给 `PQcancel` 或者 `PQfreeCancel` 。

`PQfreeCancel`

释放 `PQgetCancel` 创建的数据结构。

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` 释放一个由前面的 `PQgetCancel` 创建的数据对象。

`PQcancel`

要求服务器放弃处理当前命令。

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

如果取消请求成功发送，则返回值为 1，否则为 0。如果不成功，则 `errbuf` 里面会填充解释的错误信息。 `errbuf` 必须是一个大小为 `errbufsize` 的 char 数组（建议大小为 256 字节）。

不过，成功发送取消请求并不保证请求会有任何效果。如果取消生效， 那么当前的命令将提前结束并且返回一个错误的结果。如果取消失败（也就是说， 因为服务器已经完成命令的处理），那么就根本不会有可见的结果。

如果 `errbuf` 是信号句柄里的一个局部变量，那么 `PQcancel` 可以在一个信号句柄里安全地调用。在 `PQcancel` 涉及的范围里， `PGcancel` 对象都是只读的， 因此我们也可以从一个与处理 `PGconn` 对象的线程分离的线程里处理它。

`PQrequestCancel`

`PQrequestCancel` 是 `PQcancel` 的一个废弃的变种。

```
int PQrequestCancel(PGconn *conn);
```

要求服务器放弃对当前命令的处理。它直接在 `PGconn` 对象上进行操作， 并且如果失败，就会在 `PGconn` 对象里存储错误信息（因此可以用 `PQerrorMessage` 检索出来。）尽管功能一样，但是这个方法在多线程程序里和信号句柄里会有危险，因为它可能覆盖 `PGconn` 的错误信息，因此将可能把当前连接正在处理的操作打乱。

31.7. 捷径接口

PostgreSQL提供一个向服务器发送简单的函数调用的捷径接口。

> **Tip:** 这个接口在某种程度上已经废弃了，因为我们可以通过设置一个预备语句来定义函数调用， 从而达到类似的性能和更强大的功能。然后，用二进制参数和结果传输执行该语句，替换一次捷径函数调用。

函数 `PQfn` 请求允许通过捷径接口执行服务器函数。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

`fnid` 参数是待执行的函数的对象标识（OID）。 `args` 和 `nargs` 定义了要传递给函数的参数；它们必须匹配已经声明了的函数参数列表。如果某个参数结构的 `isint` 字段是真，那么 `u.integer` 值以指定长度（必须是1，2，或者 4 字节） 的整数发送给服务器；这时候会进行恰当的字节序交换（byte-swapping）。如果 `isint` 为假， 那么在 `*u.ptr` 里面的指定字节数的数据将不做任何处理发送出去； 这些数据必须是服务器预期的该函数参数类型的二进制传输格式。 `result_buf` 是放置返回值的缓冲区。调用者必须为返回值分配足够的空间（这里没有检查！）。 实际的返回值长度将被放在 `result_len` 指向的整数里返回。 如果预期返回值是 1，2 或 4 字节整数，把 `result_is_int` 设为 1；否则设为 0。 把 `result_is_int` 设为 1 告诉libpq必要时交换数值字节序， 这样就可以正确地传输成客户机上的 `int` 值。当 `result_is_int` 是 0 时， 服务器发送回来的字节串不做修改直接返回。

`PQfn` 总是返回一个有效的 `PGresult` 指针。 在使用结果之前应该检查结果状态。当结果不再使用后，调用者有义务使用 `PQclear` 释放 `PGresult` 。

请注意我们没办法处理空值的参数，空的结果，也没办法在使用这个接口的时候设置有值的结果。

31.8. 异步通知

PostgreSQL通过 `LISTEN` 和 `NOTIFY` 命令提供对异步通知的支持。一个客户端会话用 `LISTEN` 命令注册一个它感兴趣的通知条件 （也可以用 `UNLISTEN` 命令停止监听）。所有正在监听某一通知条件的会话在该条件名的 `NOTIFY` （通知）被任何会话执行后都将被异步地通知。 一个"payload"可以向听众传达额外的数据。

libpq应用把 `LISTEN` ， `UNLISTEN` 和 `NOTIFY` 命令作为普通的 SQL 命令提交。随后通过调用 `PQnotifies` 可以侦测到 `NOTIFY` 消息的到达。

函数 `PQnotifies` 从一个来自服务器的未处理的通知信息列表中返回下一条通知。 如果没有未处理的信息则返回 NULL 指针。一旦 `PQnotifies` 返回一条通知， 该通知会被认为已处理并且将被从通知列表中删除。

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* 通知的通道名字 */
    int  be_pid;                /* 通知服务器进程的进程ID */
    char *extra;                /* 通知负载字符串 */
} PGnotify;
```

在处理完 `PQnotifies` 返回的 `PGnotify` 对象后， 别忘了用 `PQfreemem` 把它释放。释放 `PGnotify` 指针就足够了； `relname` 和 `extra` 字段并未代表独立分配的内存。 （这些领域的名称是历史性的，尤其是频道名称与名称没有什么关系。）

Example 31-2给出了一个简单的程序，举例说明异步通知的使用。

`PQnotifies` 实际上并不读取服务器数据；它只是返回被前面的另一个 libpq函数吸收的信息。 在以前的libpq 版本里，周期性的收到 `NOTIFY` 信息的唯一方法是持续的提交命令， 即使是空查询也可以，并且在每次 `PQexec` 后检查 `PQnotifies` 。 现在这个方法也还能工作，不过我们认为它太浪费处理器时间而废弃了它。

在你没有可用的命令提交时检查 `NOTIFY` 消息的更好的方法是调用 `PQconsumeInput` ，然后检查 `PQnotifies` 。 你可以使用 `select()` 来等待服务器数据的到达， 这样在没有数据可处理时可以不浪费CPU时间。 （参阅 `PQsocket` 获取用于 `select()` 的文件描述符。） 注意这种方法不管你使用 `PQsendQuery` / `PQgetResult` 还是简单的 `PQexec` 来执行命令都能工作。不过，你应该记住在每次 `PQgetResult` 或 `PQexec` 后检查 `PQnotifies` ，看看在处理命令的过程中是否有通知到达。

31.9. 与 `COPY` 命令相关的函数

PostgreSQL里的 `COPY` 命令里有用于 libpq从网络连接读出或者写入的选项。 本节描述的函数允许应用通过提供或者消耗拷贝数据，充分利用这个功能。

整个过程是应用首先通过 `PQexec` 或者一个等效的函数发出 `COPY` 命令。对这个命令的响应（如果命令无误） 将是一个带着状态码 `PGRES_COPY_OUT` 或者 `PGRES_COPY_IN` 的 `PGresult` （具体根据声明的拷贝方向）。 应用然后就应该使用本节的函数接受或者发送数据行。在数据传输结束之后， 返回另外一个 `PGresult` 对象以表明传输的成功或者失败。它的状态将是 `PGRES_COMMAND_OK` 表示成功或者如果发生了一些问题，是 `PGRES_FATAL_ERROR` 。这个时候开始我们可以通过 `PQexec` 发出更多 SQL 命令。 （ `COPY` 操作在处理的过程中， 我们不可能用同一个连接执行其它 SQL 命令。

如果一个 `COPY` 是通过 `PQexec` 在一个可以包含额外命令的字串里发出的，那么应用在完成 `COPY` 序列之后必须继续用 `PQgetResult` 抓取结果。只有在 `PQgetResult` 返回 `NULL` 的时候，我们才能确信 `PQexec` 的命令字串已经处理完毕，并且已经可以安全地发出更多命令。

本节的这些函数应该只在从 `PQexec` 或 `PQgetResult` 获得了 `PGRES_COPY_OUT` 或 `PGRES_COPY_IN` 结果状态的情况下执行。

一个承载了这些状态值之一地 `PGresult` 对象运载了某些有关正在开始的 `COPY` 操作的额外信息。这些额外的数据可以用那些同时也处理查询结果的函数获取。

`PQnfields`

返回要拷贝的字段（数据域）个数

`PQbinaryTuples`

0 表示全部拷贝格式都是文本的（行之间用换行分隔，字段用分隔符分隔，等等）。 1 表示全部拷贝格式都是二进制。参阅COPY获取更多信息。

`PQfformat`

返回和拷贝操作的每个字段相关的格式代码（0 是文本，1 是二进制）。 如果全部拷贝格式是文本，那么每字段的格式码将总是零，但是（整体） 二进制格式可以支持文本和二进制字段并存。 （不过，就目前的 `COPY` 实现， 在二进制拷贝里只出现二进制字段；所以目前每字段的格式总是匹配整体格式。）

> **Note:** 这些额外的数据值只能在使用 3.0 版本的协议的时候获得。在使用 2.0 版本的协议时，所有这些函数都返回 0。

31.9.1. 用于发送 `COPY` 数据的函数

这些函数用于在 `COPY FROM STDIN` 过程中发送数据。 如果在连接不是处于 `COPY_IN` 状态下，它们会失败。

`PQputCopyData`

在 `COPY_IN` 状态里向服务器发送数据。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

传输指定的 `buffer` 里的，长度为 `nbytes` 的 `COPY` 数据到服务器。如果数据发送成功，结果是 1，如果因为发送企图会阻塞 （这种情况只有在连接是非阻塞模式时才有可能）而没有成功，那么是零， 或者是在发生错误的时候是 -1。 （如果返回 -1，那么使用 `PQerrorMessage` 检索细节。如果值是零，那么等待写准备好然后重试。）

应用可以把 `COPY` 数据流分隔成任意合适的大小放到缓冲区里。在发送的时候， 缓冲区的边界没有什么特殊的语意。数据流的内容必须匹配 `COPY` 命令预期的数据格式； 参阅COPY获取细节。

`PQputCopyEnd`

在 `COPY_IN` 状态里向服务器发送数据完毕的指示。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

如果 `errormsg` 是 `NULL` ，则成功结束 `COPY_IN` 操作。 如果 `errormsg` 不是 `NULL` 则 `COPY` 操作被强制失败， `errormsg` 指向的字串是错误信息。 （我们不能认为同样的信息可能会从服务器传回， 因为服务器可能已经因为自己的原因让 `COPY` 失败。 还要注意的是在使用 3.0 版本之前的协议连接时，强制失败的选项是不能用的。）

如果终止数据发送，则结果为 1，如果发送企图会阻塞（只有在连接是在非阻塞模式的情况下才可能出现这个情况）， 则为零，如果发生错误则返回 -1。 （如果返回值是 -1，用 `PQerrorMessage` 检索细节。如果值是零，那么等待写准备好然后重新尝试。）

在成功调用 `PQputCopyEnd` 之后，调用 `PQgetResult` 获取 `COPY` 命令的最终结果状态。我们可以用平常的方法来等待这个结果可用。然后返回到正常的操作。

31.9.2. 用于接收 `COPY` 数据的函数

这些函数用于在 `COPY TO STDOUT` 的过程中检索数据。 如果连接不在 `COPY_OUT` 状态，那么他们将会失败。

`PQgetCopyData`

在 `COPY_OUT` 状态下从服务器接收数据。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

在一个 `COPY` 的过程中试图从服务器获取另外一行数据。数据总是每次返回一个数据行；如果只有一部分行可用，那么它不会被返回。成功返回一个数据行包括分配一个内存块来保存这些数据。 `buffer` 参数必须是非 `NULL` 。 `*buffer` 设置为指向分配出来的内存的指针，或者是如果没有返回缓冲区，那么为 `NULL` 。 一个非 `NULL` 的结果缓冲区在不再需要的时候必须用 `PQfreemem` 释放。

在成功返回一行之后，那么返回的值就是该数据行里数据的字节数（这个将总是大于零）。返回的字串总是空结尾的，虽然可能只是对文本的 `COPY` 有用。 一个零的结果表示该 `COPY` 仍然在处理中，但是还没有可以用的行 （这个只有在 `async` 为真的时候才可能）。一个结果为 -1 的值表示 `COPY` 已经结束。结果为 -2 表示发生了错误（参考 `PQerrorMessage` 获取原因）。

在 `async` 为真的时候（非零）， `PQgetCopyData` 将不会阻塞住等待输入；如果该 `COPY` 仍在处理过程中并且没有可用的完整行，那么它将返回零。 （在这种情况下它等待读准备好，然后在再次调用 `PQgetCopyData` 之前， 调用 `PQconsumeInput` 。）在 `async` 是假（零）的时候， `PQgetCopyData` 将阻塞住，直到数据可用或者操作完成。

在 `PQgetCopyData` 返回 -1 之后，调用 `PQgetResult` 获取 `COPY` 命令的最后结果状态。我们可以用通常的方法等待这个结果可用。然后返回到正常操作。

31.9.3. 用于 `COPY` 的废弃的函数

下面的这些函数代表了以前的处理 `COPY` 的方法。尽管他们还能用，但是现在已经废弃了，因为他们的错误处理实在是太糟糕了，并且检测数据结束的方法也很不方便，并且缺少对二进制和非阻塞传输的支持。

`PQgetline`

读取一个以新行符结尾的字符行（由服务器传输）到一个长度为 `length` 的字符串缓冲区。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

这个函数拷贝最多 `length` -1个字符到缓冲区里，然后把终止的新行符转换成一个字节零。 `PQgetline` 在输入结束时返回 `EOF` ， 如果整行都被读取了返回 0， 如果缓冲区填满了而还没有遇到结束的新行符则返回 1。

注意，应用程序必须检查新行是否包含两个字符 `\.` ， 这表明服务器已经完成了 `COPY` 命令的结果的发送。 如果应用可能收到超过 `length` -1字符长的字符，我们就应该确保正确识别 `\.` 行（例如，不要把一个长的数据行的结束当作一个终止行）。

`PQgetlineAsync`

不阻塞地读取一行 `COPY` 数据（由服务器传输）到一个缓冲区中。

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

这个函数类似于 `PQgetline` ，但是可以用于那些必须异步读取 `COPY` 数据的应用，也就是不阻塞的应用。在使用了 `COPY` 命令和获取了 `PGRES_COPY_OUT` 响应之后， 应用应该调用 `PQconsumeInput` 和 `PQgetlineAsync` 直到收到数据结束的信号。

不像 `PQgetline` ， 这个函数负责检测数据结束。

在每次调用时，如果libpq的输入缓冲区内有可用的一个完整的数据行， `PQgetlineAsync` 都将返回数据。否则，在其他数据到达之前不会返回数据。 如果见到了拷贝数据结束的标志，此函数返回 -1，如果没有可用数据， 或者是给出一个正数表明返回的数据的字节数，返回 0。 如果返回 -1，调用者下一步必须调用 `PQendcopy` ，然后回到正常处理。

返回的数据将不超过一行的范围。如果可能，每次将返回一个完整行。但如果调用者提供的缓冲区太小， 无法容下服务器发出的整行，那么将返回部分行。对于文本数据，这个可以通过测试返回的最后一个字节是否是 `\n` 来确认。（在二进制 `COPY` 中，我们需要对 `COPY` 数据格式进行实际的分析，以便做相同的判断。）返回的字符串不是空结尾的。（如果你想得到一个空结尾的字串， 确保你传递了一个比实际可用空间少一字节的 `bufsize` 。）

`PQputline`

向服务器发送一个空结尾的字符串。成功时返回 0，如果不能发送字符串返回 `EOF` 。

```
int PQputline(PGconn *conn,
              const char *string);
```

一系列 `PQputline` 调用发送的 `COPY` 数据流和 `PQgetlineAsync` 返回的数据有着一样的格式， 只是应用不需要明确地在每次 `PQputline` 调用中发送一个数据行；每次调用发送多行或者部分行都是可以的。

> **Note:** 在PostgreSQL协议3.0之前，应用必须明确的发送两个字符 `\.` 作为行结束，向服务器表明它已经完成了发送 `COPY` 数据。虽然这个仍然工作， 但是已经废弃了，并且 `\.` 的特殊含义在将来的版本中有望删除。 在发送实际数据之后调用 `PQendcopy` 就足够了。

`PQputnbytes`

向服务器发送一个非空结尾的字符串。成功时返回 0，如果不能发送字符串返回 `EOF` 。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

此函数类似 `PQputline` ，除了数据缓冲区不需要是空结尾的之外， 因为要发送的字节数是直接声明的。在发送二进制数据的时候使用这个过程。

`PQendcopy`

与服务器同步。

```
int PQendcopy(PGconn *conn);
```

这个函数将等待直到服务器完成拷贝。你可以在用 `PQputline` 向服务器发送完最后一个字符串后或者用 `PGgetline` 从服务器获取最后一行字符串后调用它。我们必须调用这个函数，否则服务器可能会和前端 "不同步"。在这个函数返回后，服务器就已经准备好接收下一个 SQL 命令了。 成功时返回0，否则返回非零值。（如果返回值为非 0，用 `PQerrorMessage` 检索细节。）

在使用 `PQgetResult` 时，应用应该对 `PGRES_COPY_OUT` 的结果做出反应：重复调用 `PQgetline` ，并且在收到结束行时调用 `PQendcopy` 。然后应该返回到 `PQgetResult` 循环直到 `PQgetResult` 返回空指针。类似地， `PGRES_COPY_IN` 结果是用一系列 `PQputline` 调用最后跟着 `PQendcopy` ， 然后返回到 `PQgetResult` 循环。这样的排列将保证嵌入到一系列 SQL命令里的 `COPY` 命令将被正确执行。

旧的应用大多通过 `PQexec` 提交一个 `COPY` 命令并且假设在 `PQendcopy` 后事务完成。这样只有在 `COPY` 是命令字串里的唯一的SQL命令时才能正确工作。

31.10. 控制函数

这些函数控制许多libpq其他行为的细节。

`PQclientEncoding`

返回客户端编码。

```
int PQclientEncoding(const PGconn *_conn_);
```

请注意，它返回编码ID，而不是一个符号串如 `EUC_JP` 。 转换编码的ID为一个编码名称，你可以使用：

```
char *pg_encoding_to_char(int _encoding_id_);
```

`PQsetClientEncoding`

设置客户端编码。

```
int PQsetClientEncoding(PGconn *_conn_, const char *_encoding_);
```

`_conn_` 是一个到服务器的连接， `_encoding_` 是你想要的编码使用。如果函数成功设置编码，则返回0，否则返回-1。 这个链接的当前编码可以通过 `PQclientEncoding` 确定。

`PQsetErrorVerbosity`

决定 `PQerrorMessage` 和 `PQresultErrorMessage` 返回的信息的冗余程度。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

`PQsetErrorVerbosity` 设置冗余模式，返回连接的前一个设置。在*TERSE*模式下， 返回的消息只包括严重性，主信息，以及位置信息；这些东西通常只有一行。 缺省模式生成的消息包括上面的信息加上任何细节，提示，或者环境字段（这些可能跨越几行）。 *VERBOSE*模式包括所有可以获得的字段。修改冗余模式不会影响我们从已经存在 `PGresult` 对象中获取的信息，只有随后创建的 `PGresult` 对象才受到影响。

`PQtrace`

打开对前端/服务器通讯的跟踪，把调试信息输出到一个文件流里。

```
void PQtrace(PGconn *conn, FILE *stream);
```

> **Note:** 在Windows上，如果libpq库和应用使用了不同的标志编译， 那么这个函数调用会导致应用崩溃，因为 `FILE` 指针的内部表现形式是不一样的。 特别是多线程/单线程， 发布/调试，以及静态/动态标志应该是库和所有使用库的应用都一致。

`PQuntrace`

关闭 `PQtrace` 打开的跟踪。

```
void PQuntrace(PGconn *conn);
```

31.11. 各种函数

一如往常，也有一些函数，只是不是在任何地方都适合。

`PQfreemem`

释放libpq分配的内存。

```
void PQfreemem(void *ptr);
```

释放libpq分配的内存，尤其是 `PQescapeByteaConn` ， `PQescapeBytea` ， `PQunescapeBytea` 和 `PQnotifies` 。 尤其重要的是，在Windows系统上使用这个函数，而不是 `free()` 。 这是因为只有DLL和应用程序的多线程/单线程，发布/调试，静态/动态标志是相同的时， 才在一个DLL中分配内存，并在应用程序工作时释放内存。在非Windows平台上， 这个函数与标准库函数 `free()` 相同。

`PQconninfoFree`

释放 `PQconndefaults` 或 `PQconninfoParse` 分配的数据结构。

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

一个简单的 `PQfreemem` 不会这样做，因为数组包含对子字符串的引用。

`PQencryptPassword`

准备一个PostgreSQL密码的加密形式：

```
char * PQencryptPassword(const char *passwd, const char *user);
```

这个函数旨在用于那些发送类似于 `ALTER USER joe PASSWORD 'pwd'` 命令的客户端应用程序。 这是一个很好的方法，这种命令不发送原始的明文密码，因为它可能被暴露在命令日志，活动显示中等等。 相反，在发送前，使用这个函数可以将密码转换为加密的形式。参数是明文密码和用户的SQL名字。 返回值是 `malloc` 分配的一个字符串，或超出内存时为 `NULL` 。 调用可以认为字符串中不包含需要逃逸的特殊字符。当使用结束之后，用 `PQfreemem` 进行释放。

`PQmakeEmptyPGresult`

用给定的状态构造一个空 `PGresult` 对象。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

这是libpq的内部函数，用于分配和初始化一个空 `PGresult` 对象。 如果不能分配内存，那么这个函数返回 `NULL` 。这是输出， 因为一些应用程序发现它可以有效的生成结果对象本身（特别是带有错误状态的对象）。 如果 `conn` 非空，并且 `status` 用于表示一个错误， 那么指定连接的当前错误信息被复制到 `PGresult` 中。同时， 如果 `conn` 非空，那么连接中的任何事件过程会被复制到 `PGresult` 中。 （它们不会获得 `PGEVT_RESULTCREATE` 请求， 但会看到 `PQfireResultCreateEvents` ）。需要注意的是随着libpq 本身返回 `PGresult` 时，对象最后应该请求 `PQclear` 。

`PQfireResultCreateEvents`

为 `PGresult` 对象中的每个事件过程触发一个 `PGEVT_RESULTCREATE` 事件 （参阅Section 31.13）。成功时返回非0，如果任何事件过程失败返回0。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

`conn` 被传送给事件过程，但不会被直接使用。如果事件过程不使用它，则会返回 `NULL` 。

已经接收到这个对象的 `PGEVT_RESULTCREATE` 或 `PGEVT_RESULTCOPY` 事件的事件过程不会被再次触发。

这个函数与 `PQmakeEmptyPGResult` 分开的主要原因是它经常创建一个 `PGresult` ，并且在调用事件过程之前就用数据对其进行填充。

`PQcopyResult`

完成一个 `PGresult` 对象的拷贝。这个拷贝不会以任何方式来连接到资源结果， 并且当该拷贝不再需要时，需要调用 `PQclear` 进行清理。如果函数失败，返回 `NULL` 。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

不会制作一个明确的拷贝。返回的结果通常会是 `PGRES_TUPLES_OK` 状态， 并且不会拷贝资源中的错误信息，然而会拷贝命令状态字符串。 `flags` 决定其他需要拷贝的。通常是几个 `PG_COPYRES_ATTRS` 的按位或。 `PG_COPYRES_ATTRS` 声明复制源结果的属性（列定义）。 `PG_COPYRES_TUPLES` 声明复制源结果的元组（这意味着也复制属性）。 `PG_COPYRES_NOTICEHOOKS` 声明复制源结果的通知陷阱。 `PG_COPYRES_EVENTS` 声明负值源结果的事件。 （但任何与源关联的实例数据不会被复制。）

`PQsetResultAttrs`

设置 `PGresult` 对象的属性。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

提供的 `attDescs` 被复制到结果中。如果 `attDescs` 指针为 `NULL` ，或 `numAttributes` 小于1，那么请求将被忽略， 并且函数成功。如果 `res` 已经有了属性，那么函数会失败。如果函数失败， 会返回0。如果函数成功，会返回非0。

`PQsetvalue`

设置 `PGresult` 对象的元组字段值。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

这个函数会自动按需增加结果的内置元组。然而， `tup_num` 参数必须小于等于 `PQntuples` ，意味着这个函数一次只能增加一个元组。 但已存在的任意的元组中的任意字段可以以任意顺序进行调整。如果 `field_num` 中的一个值已经存在，会被覆盖重写。如果 `len` 是-1，或 `value` 是 `NULL` ，字段值会被设置为一个SQL空值。 `value` 被复制到结果的私有存储中，因此函数返回结果后就不再需要了。如果函数失败，会返回0。 如果函数成功，会返回非0。

`PQresultAlloc`

为 `PGresult` 对象分配子存储。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

当 `res` 被清理时，该函数分配的内存也会被释放掉。如果函数失败， 返回 `NULL` 。结果是保证任何类型的数据能够充分对齐，如同对 `malloc` 一样。

`PQlibVersion`

返回正在使用的libpq的版本。

```
int PQlibVersion(void);
```

如果特定的功能在libpq当前加载的版本中可用，那么用于决定运行时此函数的结果。 该函数可以使用，比如，用来确定可用于 `PQconnectdb` 的连接选项， 或者是否支持PostgreSQL 9.0中添加的 `hex` `bytea` 输出。

数字是通过把主、次及版本号转换成两位十进制数并且把它们连接在一起组成的。例如， 版本9.1将被返回901000，版本9.1.2将被返回90102（前导零没有显示）。

> **Note:** 这个函数是在PostgreSQL版本9.1中出现的，所以它不能用来在较早的版本中检测所需功能， 因此连接它将在版本9.1上创建一个连接依赖。

31.12. 注意信息处理

服务器生成的注意信息和警告信息都不会由查询执行函数返回，因为他们并不蕴涵着查询的失败。 它们会被传递给一个注意信息处理函数，然后在该处理返回之后继续正常执行。 缺省的注意信息处理函数在 `stderr` 上打印该信息， 但是应用可以通过提供自己的处理函数来覆盖这个行为。

由于历史原因，系统里存在两个级别的注意信息处理，分别叫做注意信息接收器和注意信息处理器。 缺省的行为是注意信息接收器格式化注意信息然后传递给注意信息处理器一个字串进行打印。 不过，对于自行处理这些事情的应用而言，通常是忽略注意信息处理器层，而只是在注意信息接收器里完成所有动作。

函数 `PQsetNoticeReceiver` 为一个连接对象设置或者检查当前的注意信息接收器。 类似的是 `PQsetNoticeProcessor` 设置或者检查当前的注意信息处理器。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

这些函数都返回前一个注意信息接收器或者处理器函数指针，然后设置新的数值。 如果你提供一个空函数指针，那么就不会执行任何动作，但是返回当前指针。

当我们从服务器获取一个注意或者警告信息的时候，或者是收到libpq 内部生成的类似信息时，注意信息接收器函数将被调用。消息会以一个 `PGRES_NONFATAL_ERROR` 的 `PGresult` 的形式传递。（这就允许接收器用 `PQresultErrorField` 抽取独立的字段，或者用 `PQresultErrorMessage` 完成预先格式化好的信息。） 传递给 `PQsetNoticeReceiver` 的同一个 void 指针也同样传递给该函数。 （必要时，这个指针可以用来访问应用相关的状态。）

缺省的注意信息接收器只是简单的抽取信息（使用 `PQresultErrorMessage` ） 然后传递给注意信息处理器。

注意信息处理器负责处理一个以文本形式给出的注意或者警告信息。系统传递给他消息的字串文本 （包括结尾的新行符），加上一个和传递给 `PQsetNoticeProcessor` 一样的 void （无类型）指针。（必要时，这个指针可以用来访问应用相关的状态。）

缺省的注意信息处理器就是：

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

一旦你设置了注意消息接收器或者处理器，那么你就应该准备好在 `PGconn` 对象或者 `PGresult` 对象开始存在的时候起就有人调用它们。在创建 `PGresult` 的时候， `PGconn` 的当前注意信息处理指针被拷贝到 `PGresult` ，以便被类似 `PQgetvalue` 这样的函数使用。

31.13. 事件系统

libpq事件系统用于通知对libpq 事件感兴趣的注册事件处理过程，如创建或删除 `PGconn` 和 `PGresult` 对象。一个主要的使用原因是，它允许应用程序通过一个 `PGconn` 或 `PGresult` 关联它们自己的数据， 并且确保数据在适当的时候释放。

每个注册的事件处理程序与两片数据相关联，已知的libpq 只作为不透明的 `void *` 指针。当事件处理程序注册带有 `PGconn` 时， 应用程序会提供一个*passthrough*指针。传递指针在由它产生的 `PGconn` 和所有的 `PGresult` 的生命周期中永远不会改变，它指向生命周期长的数据。除此之外，还有一个*instance data*指针，它从每个 `PGconn` 和 `PGresult` 中的 `NULL` 开始。这个指针可以与 `PQinstanceData` ， `PQsetInstanceData` ， `PQresultInstanceData` 和 `PQsetResultInstanceData` 函数一起使用。需要注意的是不同于传递指针， 一个 `PGconn` 的实例数据不会被由它产生的 `PGresult` 自动继承。 libpq不知道传递和实例数据指针指向的是什么，并且不会尝试去释放它们；这对事件处理程序是一种保证。

31.13.1. 事件类型

枚举 `PGEventId` 命名事件系统处理的事件的类型。所有的命名值都是从 `PGEVT` 开始。对每个事件类型来说，有一个相应的事件信息结构， 用于传送传递给事件处理程序的参数。事件类型如下：

`PGEVT_REGISTER`

当调用 `PQregisterEventProc` 时，会发生注册的事件。这是一个理想化的时间， 用于初始化任意 `instanceData` ，可能需要一个事件过程。 每次连接中的每个事件处理程序只会触发一个注册了的事件。如果事件过程失败，会终止注册。

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

当接收到 `PGEVT_REGISTER` 时， `evtInfo` 指针应该被转换为一个 `PGEventRegister *` 。 这个结构包含了一个 `CONNECTION_OK` 状态的 `PGconn` ；用以保证在获得一个好的 `PGconn` 之后立即请求调用 `PQregisterEventProc` 。当返回一个错误代码时，必须执行所有的清理， 因为没有 `PGEVT_CONNDESTROY` 会被发送。

`PGEVT_CONNRESET`

`PQreset` 或 `PQresetPoll` 函数完成时，触发连接复位事件。 在这两种情况下，只有重置成功时才会触发事件。如果事件过程失败，整个连接复位都会失败； `PGconn` 被置为 `CONNECTION_BAD` 状态并且 `PQresetPoll` 将返回 `PGRES_POLLING_FAILED` 。

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

当接收到一个 `PGEVT_CONNRESET` 事件时， `evtInfo` 指针应该被转换为一个 `PGEventConnReset *` 。尽管包含的 `PGconn` 被重置了，但所有事件数据不会改变。 这个事件应该用于reset/reload/requery任何关联的 `instanceData` 。 需要注意的是即使事件过程在处理 `PGEVT_CONNRESET` 时失败了，当连接关闭时， 仍会接收一个 `PGEVT_CONNDESTROY` 事件。

`PGEVT_CONNDESTROY`

在响应 `PQfinish` 时会触发连接破坏事件。这是事件过程的职责： 合适的清理它的事件数据，因为libpq没有能力管理这部分内存。失败的清理会导致内存溢出。

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

当接收到一个 `PGEVT_CONNDESTROY` 事件时， `evtInfo` 指针应该被转换为一个 `PGEventConnDestroy *` 。 在 `PQfinish` 执行清理之前会触发该事件。事件过程的返回值会被忽略， 因为没有很好的方式从 `PQfinish` 指出失败。同样， 一个事件过程失败不应该中止清理不需要的内存的过程。

`PGEVT_RESULTCREATE`

回应任意产生一个结果（包括 `PQgetResult` ）的查询执行函数时， 会触发结果创建事件。这个事件只有在成功创建结果时才会被触发。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

当接收到一个 `PGEVT_RESULTCREATE` 事件时， `evtInfo` 指针应该被转换为一个 `PGEventResultCreate *` 。 `conn` 是用于产生结果的连接。这是理想的位置，用于初始化任意需要与结果相关联的 `instanceData` 。 如果事件过程失败，结果会被清理并且传播该失败。事件过程不应该尝试自己 `PQclear` 结果对象。 当返回一个错误代码时，必须执行所有的清理，因为没有 `PGEVT_RESULTDESTROY` 会被发送。

`PGEVT_RESULTCOPY`

在响应 `PQcopyResult` 时会触发结果拷贝事件。只有在拷贝完成时， 才会触发该事件。只有那些为源结果成功处理 `PGEVT_RESULTCREATE` 或 `PGEVT_RESULTCOPY` 事件的事件过程才会收到 `PGEVT_RESULTCOPY` 事件。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

当接收到一个 `PGEVT_RESULTCOPY` 事件时， `evtInfo` 指针应该被转换为一个 `PGEventResultCopy *` 。 `src` 结果是当 `dest` 为拷贝目标时要进行拷贝的。这个事件用于提供一个 `instanceData` 的深度拷贝，因为 `PQcopyResult` 做不到。 如果事件过程失败，整个拷贝过程都将失败，并且 `dest` 结果也会被清理。 当返回一个错误代码时，必须执行所有的清理，因为没有 `PGEVT_RESULTDESTROY` 事件会被发送。

`PGEVT_RESULTDESTROY`

在回应 `PQclear` 时会触发结果破坏事件。这是事件过程的责任； 合理清理它的事件数据，因为libpq没有能力管理这块内存。清理失败会导致内存溢出。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

当接收到一个 `PGEVT_RESULTDESTROY` 事件时， `evtInfo` 指针应该被转换为一个 `PGEventResultDestroy *` 。 这个事件会在 `PQclear` 执行清理之前被触发。事件过程的返回结果会被忽略， 因为没有一个方式能够从 `PQclear` 指出失败。同样， 一个事件过程失败不应该中止对不需要内存的清理。

31.13.2. 事件回调过程

`PGEventProc`

`PGEventProc` 是一个事件过程中指针的typedef，也就是， 从libpq接收事件的用户回调函数。 事件过程的用法必须如下：

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

`evtId` 参数指出要发生哪个 `PGEVT` 事件。 `evtInfo` 指针必须被转换为合适的结构类型以获取有关该事件的进一步信息。 `passThrough` 是当事件过程被注册时，提供给 `PQregisterEventProc` 的指针。这个函数应该返回一个非0的值，如果成功的话，反之，返回0。

在任意 `PGconn` 中，一个特殊的事件过程只能注册一次。 这是因为过程地址被用于作为查询关键字，以识别相关的实例数据。

Caution

在Windows上，函数可以有两个不同的地址：一个是内部DLL可见的，另一个是外部DLL可见的。 需要注意的是，libpq事件过程函数只会使用其中一个地址，否则会造成混乱。有效的编写代码的简单规则是为了保证事件过程声明为 `static` 。 如果过程地址在它的源文件之外是可用的，公开一个单独的函数以返回地址。

31.13.3. 事件支持函数

`PQregisterEventProc`

用libpq注册一个事件回调过程。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

在每个 `PGconn` 中必须注册一次事件过程，用于希望接受到的事件。除了内存之外， 对于一次连接注册的事件过程个数没有限制。如果成功，则返回一个非0的值，否则返回0。

当一个libpq事件被触发时，会调用一个 `proc` 参数。 内存地址同样会被用于查找 `instanceData` 。 `name` 用于指出在错误信息中的事件过程。这个值不能为 `NULL` 或一个长度为0的字符串。 名字字符串被拷贝到 `PGconn` 中，因此被传递的不需要拥有很长的生命周期。 `passThrough` 指针被传递到 `proc` ， 不管何时触发事件。这个参数可以是 `NULL` 。

`PQsetInstanceData`

为 `proc` 到 `data` 的过程设置连接 `conn` 的 `instanceData` 。成功则返回一个非0值，否则返回0。 只有 `proc` 没有成功在 `conn` 注册时才会发生失败。

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

`PQinstanceData`

返回与 `proc` 过程，或 `NULL` （如果存在空） 相关的 `conn` 的 `instanceData` 。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

为 `proc` 到 `data` 的过程设置结果的 `instanceData` 。 成功则返回一个非0值，否则返回0。只有 `proc` 没有成功在结果注册时才会发生失败。

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

`PQresultInstanceData`

返回与 `proc` 过程，或 `NULL` （如果存在空）相关的结果的 `instanceData` 。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

31.13.4. 事件例子

一个管理与libpq连接和结果相关的私有数据的例子：

```
/* <!-- required header for libpq events (note: includes libpq-fe.h) -->需要libpq事件的头文
#include <libpq-events.h>

/* The instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn = PQconnectdb("dbname = postgres");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    <!--
/* called once on any connection that should receive events.
     * Sends a PGEVT_REGISTER to myEventProc.
     */
 -->
 /* 在任何应该接收事件的连接上调用一次。
     * 发送一个 PGEVT_REGISTER 到 myEventProc。
     */

    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    <!-- /* conn instanceData is available */ -->
/* conn instanceData 是可用的 */
    data = PQinstanceData(conn, myEventProc);

    <!-- /* Sends a PGEVT_RESULTCREATE to myEventProc */ -->
/* 发送一个 PGEVT_RESULTCREATE 到 myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    <!-- /* result instanceData is available */ -->
/* 结果 instanceData 是可用的 */
    data = PQresultInstanceData(res, myEventProc);

    <!-- /* If PG_COPYRES_EVENTS is used, sends a PGEVT_RESULTCOPY to myEventProc */ -->
```

```
 /* 如果使用了 PG_COPYRES_EVENTS，发送一个 PGEVT_RESULTCOPY到 myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    <!--
/* result instanceData is available if PG_COPYRES_EVENTS was
     * used during the PQcopyResult call.
     */
 -->
 /* 结果 instanceData 是可用的，如果 PG_COPYRES_EVENTS
     * 在PQcopyResult调用期间使用了的话。
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    <!-- /* Both clears send a PGEVT_RESULTDESTROY to myEventProc */ -->
/* 两个clears都发送 PGEVT_RESULTDESTROY 到 myEventProc */
    PQclear(res);
    PQclear(res_copy);

    <!-- /* Sends a PGEVT_CONNDESTROY to myEventProc */ -->
/* 发送一个 PGEVT_CONNDESTROY 到 myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            <!-- /* associate app specific data with connection */ -->
/* 将应用程序特定的数据与连接相关联 */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            <!-- /* free instance data because the conn is being destroyed */ -->
/* 释放实例数据，因为conn被破坏了 */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            <!-- /* associate app specific data with result (copy it from conn) */ -->
/* 将应用程序特定的数据与结果相关联 (从conn中拷贝) */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
```

```
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            <!-- /* associate app specific data with result (copy it from a result) */ --
/*将应用程序特定的数据与结果相关联 (从结果中拷贝)*/
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            <!-- /* free instance data because the result is being destroyed */ -->
/* 释放实例数据，因为结果被破坏了 */
            if (data)
              free_mydata(data);
            break;
        }

        <!-- /* unknown event ID, just return TRUE. */ -->
/* 未知事件ID，只是返回TRUE。*/
        default:
            break;
    }

    return TRUE; /* <!-- event processing succeeded -->事件处理成功 */
}
```

31.14. 环境变量

下面的环境变量可以用于选择缺省的连接参数值，这些值将被 `PQconnectdb` ，`PQsetdbLogin` 和 `PQsetdb` 使用，如果调用代码没有直接声明相应值的话。比如，这些（环境变量）可以避免把硬编码数据库连接信息写入简单的客户端应用中。

- `PGHOST` 与host连接参数表现行为相同。
- `PGHOSTADDR` 与hostaddr连接参数表现行为相同。这样可以代替或除去PGHOST以避免DNS查找的开销。
- `PGPORT` 与port连接参数表现行为相同。
- `PGDATABASE` 与dbname连接参数表现行为相同。
- `PGUSER` 与user连接参数表现行为相同。
- `PGPASSWORD` 与password连接参数表现行为相同。出于安全原因是不推荐使用这个环境变量的，因为某些操作系统允许非root用户通过ps 看到进程的环境变量，而是考虑使用 `~/.pgpass` 的文件（详情请看Section 31.15）。
- `PGPASSFILE` 指定密码文件的名称用于查找。如果没有设置，默认为 `~/.pgpass` （见Section 31.15）。
- `PGSERVICE` 与service连接参数表现行为相同。
- `PGSERVICEFILE` 指定连接服务的文件中每个用户的名字，如果没有设置默认 `~/.pg_service.conf` (见Section 31.16)。
- `PGREALM` 设置与PostgreSQL一起使用的 Kerberos 域，如果该域与本地域不同的话。如果设置了 `PGREALM` ，libpq 应用将试图用这个域（realm）与服务器进行认证并且使用独立的门票文件（ticket files）以避免与本地的门票文件冲突。只有在服务器选择了Kerberos 认证时才使用这个环境变量。
- `PGOPTIONS` 与options连接参数表现行为相同。
- `PGAPPNAME` 与application_name连接参数表现行为相同。
- `PGSSLMODE` 与sslmode连接参数表现行为相同。
- `PGREQUIRESSL` 与requiressl连接参数表现行为相同。
- `PGSSLCOMPRESSION` 与sslcompression连接参数表现行为相同。
- `PGSSLCERT` 与sslcert连接参数表现行为相同。
- `PGSSLKEY` 与sslkey连接参数表现行为相同。

- `PGSSLROOTCERT` 与sslrootcert连接参数表现行为相同。
- `PGSSLCRL` 与sslcrl连接参数表现行为相同。
- `PGREQUIREPEER` 与requirepeer连接参数表现行为相同。
- `PGKRBSRVNAME` 与krbsrvname连接参数表现行为相同。
- `PGGSSLIB` 与gsslib连接参数表现行为相同。
- `PGCONNECT_TIMEOUT` 与connect_timeout连接参数表现行为相同。
- `PGCLIENTENCODING` 与client_encoding连接参数表现行为相同。

下面的环境变量可以用于为每个PostgreSQL会话声明缺省特性。（又见ALTER ROLE和ALTER DATABASE命令，在每用户或每数据库的基础上设置缺省行为。）

- `PGDATESTYLE` 设置缺省的日期/时间表现形式。（等效于 `SET datestyle TO ...`。）
- `PGTZ` 设置缺省的时区。（等效于 `SET timezone TO ...`。）
- `PGGEQO` 为基因优化器设置缺省模式。（等效于 `SET geqo TO ...`。）

请参考SQL命令SET获取这些环境变量的正确数值。

下面的环境变量决定libpq的内部行为；它们覆盖编译的缺省。

- `PGSYSCONFDIR` 设置包含 `pg_service.conf` 文件和未来版本中可能的其他系统范围的配置文件的目录。
- `PGLOCALEDIR` 设置包含信息国际化的 `locale` 文件目录。

31.15. 口令文件

用户家目录中的 `.pgpass` 或者 `PGPASSFILE` 引用的文件是一个可以包含口令的文件。如果连接要求口令（并且没有用其它方法声明口令）， 那么可以用它。 在 Microsoft Windows 上，文件名字是 `%APPDATA%\postgresql\pgpass.conf` （ `%APPDATA%` 指用户配置里的 Application Data 子目录）。

这个文件应该有下面这样的格式行：

```
_hostname_:_port_:_database_:_username_:_password_
```

你可以通过复制上面的行并且在前面添加 `#` 用于添加提醒注释到文件， 头四个字段每个都可以是一个文本值，或者 `*` ，它匹配所有的东西。 第一个匹配当前连接参数的口令行的口令域将得以使用。（因此，如果你使用了通配符， 那么应该把最具体的记录放在前面。）如果记录包含 `:` 或者 `\` ， 应该用 `\` 逃逸。一个 `localhost` 的主机名匹配来自本机的 TCP（主机名 `localhost` ）和Unix域套接字（ `pghost` 为空或缺省的套接字目录）连接。 在备用服务器中，一个名字为 `replication` 的数据库被匹配到主服务器的流复制连接。 `database` 字段用处有限，因为对同一集群中所有数据库用户有相同密码。

在linux系统中， `.pgpass` 的权限必须不允许任何全局或者同组的用户访问； 我们可以用命令 `chmod 0600 ~/.pgpass` 实现这个目的。如果权限比这个松， 这个文件将被忽略。在 Microsoft Windows上，假定该文件存储在一个安全的目录中，所以没有做特殊的权限检查。

31.16. 连接服务的文件

连接服务文件允许libpq连接参数可以与一个单一的服务名称相关联。 服务名之后可以被一个libpq连接声明，并且会使用相关设置。 允许连接参数无需要求一个重新编译的libpq应用进行修改。 服务名同样可以用 `PGSERVICE` 环境变量进行声明。

连接服务文件可以是在 `~/.pg_service.conf` ， 或者 `PGSERVICEFILE` 环境变量声明位置中的一个按用户的服务文件，也可以是一个位于 `etc/pg_service.conf` ， 或者 `PGSYSCONFDIR` 环境变量声明位置中的一个全系统文件。 如果定义的服务名与在用户和系统文件中的名字相同，那么用户文件优先。

该文件使用"INI file"格式（章节名是服务名，并且参数是连接参数）；参阅Section 31.1.2获取一个列表，例如：

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

在 `share/pg_service.conf.sample` 中提供了一个示例文件。

31.17. LDAP查找连接参数

如果libpq已经通过LDAP支持（`configure` 的 `--with-ldap`）进行了编译，可以从一个中央服务器，通过LDAP检索连接选项，如 `host` 或 `dbname` 。这样做的好处是，如果一个数据库连接参数发生了改变，在所有客户端的连接信息不必进行改变。

LDAP连接参数查找使用连接服务文件 `pg_service.conf` (参阅 Section 31.16)。在 `pg_service.conf` 中的以 `ldap://` 开始的一行被看做是一个LDAP URL，并且会执行一个LDAP查询。返回结果会是一个 `keyword = value` 的列表，用于设置连接选项。URL必须符合RFC 1959，并且是如下形式：

```
ldap://[_hostname_[:_port_]]/_search_base_?_attribute_?_search_scope_?_filter_
```

这里 `_hostname_` 缺省为 `localhost` 和 `_port_` 缺省为389。

`pg_service.conf` 的处理在LDAP成功查找之后就会被终止，但如果不能成功连接LDAP服务，那么会继续。这是为了进一步指向不同的LDAP服务器的LDAP URL线而提供的一个回滚，标准的 `keyword = value` 对格式，或缺省的连接参数。如果想在这种情况下获得一个错误信息，可以在LDAP URL后添加一个语法不正确的行。

LDIF文件创建的一个样本LDAP条目：

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn:mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

可能被下列的 LDAP URL查询：

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)
```

也可以通过LDAP查找来混合日常服务文件。一个 `pg_service.conf` 中完整的一节的例子如下：

```
# only host and port are stored in LDAP, specify dbname and user explicitly
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

31.18. SSL 支持

PostgreSQL本机支持使用SSL连接对客户端/服务器通讯进行加密， 以增强安全性。参阅Section 17.9获取服务器端SSL功能的细节。

libpq读取全系统OpenSSL配置文件， 默认情况下，文件命名为 `openssl.cnf` 并且存放在 `openssl version -d` 报告的目录中。此默认可以通过设置环境变量 `OPENSSL_CONF` 为所需配置文件的名称来重写。

31.18.1. 服务器证书的客户端验证

缺省，PostgreSQL不会执行任何服务器证书验证。 这就意味着可以在客户端没有察觉的情况下骗过服务认证（如，通过修改一个DNS记录或接管服务IP地址）。 为了避免这种情况，必须使用SSL证书认证。

如果 `sslmode` 参数设置为 `verify-ca` ， libpq将通过检查受信任的证书颁发机构的证书链(CA)来验证服务是可信任的。 如果 `sslmode` 设置为 `verify-full` ， libpq也会通过验证服务主机名匹配认证来认为服务是可信任的。 如果服务验证不能被通过，那么SSL连接会失败。在大多数对安全要求较高的环境中， 建议使用 `verify-full` 。

在 `verify-full` 模式下，认证的 `cn` (Common Name)属性与主机名进行匹配。 如果 `cn` 以 `*` 开始，会被看做是一个通配符，并且会匹配除了 点（ `.` ）之外的所有字符。这就意味着认证不会匹配子域名。 如果是使用IP而不是主机名进行连接，会进行IP匹配检查（不会做DNS检查）。

为了允许服务器认证通过，一个或多个信任的CA认证必须放在用户的home目录下的 `~/.postgresql/root.crt` 文件中。Windows下的文件名是 `%APPDATA%\postgresql\root.crt` 。

如果存在 `~/.postgresql/root.crl` 文件（Windows下是 `%APPDATA%\postgresql\root.crl` 文件），同样也会检查证书吊销列表（CRL）。

root认证文件和CRL的位置可以通过设置 `sslrootcert` 和 `sslcrl` 连接参数，或 `PGSSLROOTCERT` 和 `PGSSLCRL` 环境变量进行修改。

> **Note:** 为了与早期PostgreSQL版本兼容，如果存在一个根CA文件， `sslmode` = `require` 的行为将和 `verify-ca` 表现的相同，意味着服务器证书是经过CA验证的。 不建议依赖于这个行为，需要证书验证的应用应该总是使用 `verify-ca` 或 `verify-full` 。

31.18.2. 客户端证书

如果服务器要求一个信任的客户端认证，libpq将发送存储在用户home目录中 `~/.postgresql/postgresql.crt` 文件中的证书。该证书必须由服务器信任的证书认证 （CA）之一签名。同时也必须出示一个匹配的私钥文件 `~/.postgresql/postgresql.key` 。私钥文件不允许任何对世界或组的访问； 通过 `chmod 0600 ~/.postgresql/postgresql.key` 命令可以实现。在Windows上，这个文件是 `%APPDATA%\postgresql\postgresql.crt` 和 `%APPDATA%\postgresql\postgresql.key` ，同时没有特定的权限检查， 因为目录被认为是安全的。证书和key文件的位置可以通过 `sslcert` 和 `sslkey` 连接参数，或 `PGSSLCERT` 和 `PGSSLKEY` 环境变量进行覆盖重写。

在一些情况下，客户端认证可能会被一个"intermediate"的证书认证来签名， 而不是一个能直接被服务信任的。为了使用一个这种认证，向 `postgresql.crt` 文件追加证书签字权，并且直到"root"都可以被服务器信任。 root认证应该被包含在任何一种情况（ `postgresql.crt` 包含多个认证）下。

需要注意的是， `root.crt` 列出了最高级别的CA， 认为对签名服务证书来说是可信任的。原则上，不需要列出签名客户端认证的CA， 尽管在大多数情况下，CA仍会被认为对服务器认证是可信任的。

31.18.3. 在不同的模式提供保护

`sslmode` 参数的不同值提供了不同的保护级别。SSL可以为三种攻击提供保护：

Eavesdropping（窃听）

如果一个第三方可以在客户端与服务器端之间检查网络通信，那么它就能读取两边的连接信息 （包括用户名和密码）以及传递的数据。对此，SSL通过加密进行防护。

裁判(MITM)

如果客户端和服务器端进行传递数据的时候，第三方可以对其进行修改，那么他就能伪装成服务器， 然后查看或修改数据（即使是加密的）。第三方接着可以向原始服务器发出连接信息和数据， 最终造成无法防护这种攻击。这种攻击常用的载体有DNS中毒或IP绑架， 即客户端被定向到预期之外的不同的服务器。同样还有几种其他的方法也能做到这种攻击。 SSL通过服务器到客户端的证书验证来阻止这种攻击。

Impersonation（模拟）

如果第三方可以伪装成一个认证了的客户端，那么它就能轻松访问到它本来不能访问的数据。 典型的，如不安全的密钥管理，就会造成这种情况。SSL 通过客户端认证来阻止这种情况，即确保只有知道有效认证的人员才能访问连接服务器。

对于一个被称为安全的连接来说，进行连接之前，必须在客户端和服务器端都进行SSL配置。如果只在服务器端进行配置，在它知道服务器端需要高级认证之前不会发送敏感信息（如密码等）。 在libpq中，可以通过将 `sslmode` 参数设置为 `verify-full` 或 `verify-ca` 来确保安全

连接，并且为系统提供一个root认证以进行安全认证。类似于使用 `https` 和URL进行加密网页浏览。

一旦服务器已经认证，客户端就可以发送敏感信息。这就意味着直到这一刻，客户端都不需要知道， 是否认证需要证书，只在服务器配置，对其安全地指定。

所有以加密和密钥交换方式得SSL选项都会产生开销，因此在性能和安全之间需要进行一个权衡。 Table 31-1说明不同 `sslmode` 值的安全风险， 以及关于安全和开销所做出的声明：

Table 31-1. SSL 模式说明

`sslmode`	窃听保护	MITM保护	声明
`disable`	否	否	我不关心安全，我不想来支付加密的开销。
`allow`	可能	否	我不关心安全性，但我会支付的加密开销，如果服务器的坚持的话。
`prefer`	可能	否	我不关心加密，但我想支付加密开销，如果服务器支持它。
`require`	是	否	我希望我的数据加密，我接受开销。我相信该网络将确保我始终连接到我想要的服务器。
`verify-ca`	是	`取决于CA` 的政策	我希望我的数据加密，我接受开销。我想要确保我连接到了一个我信任的服务器。
`verify-full`	是	是	我希望我的数据加密，我接受开销。我想要确保我连接到了一个我信任的服务器， 并且是我指定的那个服务器。

`verify-ca` 和 `verify-full` 之间的不同是根据root CA的政策。 如果使用的是一个公用CA， `verify-ca` 允许那些带有CA 注册的客户端对服务器进行连接访问。在这种情况下， 应该使用 `verify-full` 。 如果使用的是一个本地CA，甚至是一个自签名证书， 使用 `verify-ca` 通常会提供充分的保护。

`sslmode` 缺省值是 `prefer` 。如在表中说明的那样， 从安全角度来看这样做是没有意义的，并且如果可能的话，它只承诺性能的开销。 它仅提供了缺省向后兼容性，在安全部署中不建议使用。

31.18.4. SSL 客户端文件的使用

Table 31-2总结了与客户端SSL设置相关的文件。

Table 31-2. libpq/客户端SSL文件的使用

文件	内容	影响
`~/.postgresql/postgresql.crt`	客户端证书	服务器要求的
`~/.postgresql/postgresql.key`	客户端的私钥	证明由所有者发送的客户端证书，并不表示证书拥有者是值得信赖的
`~/.postgresql/root.crt`	受信任的证书颁发机构	检查服务器证书是由受信任的证书机关签署。
`~/.postgresql/root.crl`	证书颁发机构吊销证书	服务器证书必须不在这个名单

31.18.5. SSL 库初始化

如果应用程序初始化 `libssl` 和/或 `libssl` 库以及libpq 编译为支持SSL，应该调用 `PQinitOpenSSL` 来告诉libpq 说 `libssl` 和/或 `libcrypto` 库已经被应用程序初始化了， 因此libpq将不会再初始化这些库。参阅 http://h71000.www7.hp.com/doc/83final/BA554_90007/ch04.html 获取关于SSL API的详细信息。

`PQinitOpenSSL`

允许应用程序选择安全库初始化。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

当 `do_ssl` 为非0时，在第一次打开一个数据库连接之前，libpq 将初始化OpenSSL库。当 `do_crypto` 为非0时， `libcrypto` 库将被初始化。缺省(如果 `PQinitOpenSSL` 没有被调用)，两个库都会被初始化。 如果没有编译SSL支持，会提供该函数，但不会做任何事情。

如果应用程序使用并初始化OpenSSL，或其底层 `libcrypto` 库， 那么在第一次打开一个数据库连接之前，必须调用这个函数（带有适当0值的参数）。 同样要确保在打开一个数据库连接之前做过初始化。

`PQinitSSL`

允许应用程序选择初始化哪个安全库。

```
void PQinitSSL(int do_ssl);
```

此功能相当于 `PQinitOpenSSL(do_ssl, do_ssl)` 。 它的应用是足够的同时初始化或都不初始化OpenSSL和 `libcrypto` 。

`PQinitSSL` 在PostgreSQL 8.0就已经出现了， 而 `PQinitOpenSSL` 是在PostgreSQL 8.4添加进来的。 所以对老版本的libpq的使用， `PQinitSSL` 是对应用程序的不错的选择。

31.19. 在多线程程序里的行为

libpq是可重入的并且是线程安全的。另外，在你编译自己的应用代码时，可能需要使用额外的编译器命令行选项。参考你的系统的文档获取关于如果建立线程可用的应用的信息，或在 `src/Makefile.global` 中查看 `PTHREAD_CFLAGS` 和 `PTHREAD_LIBS` 。此功能允许查询libpq的线程安全状态：

`PQisthreadsafe`

返回libpq库的线程安全状态。

```
int PQisthreadsafe();
```

libpq是线程安全的时，返回1 ；如果不是返回0。

一个线程限制是，两个线程不能试图同时操作同一个 `PGconn` 对象。特别是，你不能从不同的线程里通过同一个连接对象发出并发的命令。（如果你需要运行并行命令，请使用多个连接。）

`PGresult` 对象在创建后是只读的，因此可以自由地在线程之间传递。然而，如果你使用在Section 31.11和Section 31.13 中描述的任何 `PGresult` 修改函数，那么它也取决于你可以避免同一 `PGresult` 上的并发操作。

过时了的函数 `PQrequestCancel` 和 `PQoidStatus` 都是线程不安全的，因此不应该在一个多线程的程序里面使用。`PQrequestCancel` 可以由 `PQcancel` 代替。`PQoidStatus` 可以由 `PQoidValue` 代替。

如果在你的应用内部使用了 Kerberos （而不仅仅是libpq里面），你就需要在 Kerberos 调用周围锁住，因为 Kerberos 函数不是线程安全的。参阅libpq源代码里面的 `PQregisterThreadLock` 获取一个在libpq和你的应用之间进行恰当锁定的方法。

如果你的线程应用有问题，那么运行一个在 `src/tools/thread` 里的程序，看看你的平台是否有线程不安全的函数。这个程序由 `configure` 运行，但如果是二进制版本，你的库可能就不能和制作二进制的那个库匹配了。

31.20. 制作libpq程序

要制作（也就是说编译和链接）你的libpq程序， 你需要做下面的一些事情：

- 包含 `libpq-fe.h` 头文件：

  ```
  #include &lt;libpq-fe.h&gt;
  ```

  如果你没干这件事，那么你通常会看到类似下面这样的来自编译器的错误信息:

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- 告诉你的编译器PostgreSQL头文件的安装位置，方法是给你的编译器提供 ``` -I``_directory_ ``` 选项。(有些时候编译器会查找缺省的目录， 因此你可以忽略这些选项。) 比如，你的编译命令行看起来像：

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  如果你在使用制作文件(makefile)，那么向 `CPPFLAGS` 变量中增加下面的选项：

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  如果你的程序可能会被别人编译，那么你应该避免像上面那样把目录路径写成硬编码。取而代之的是你可以运行 `pg_config` 工具找出头文件在系统的什么地方：

  ```
  &lt;samp class="literal"&gt;$&lt;/samp&gt; pg_config --includedir
  &lt;samp class="literal"&gt;/usr/local/include&lt;/samp&gt;
  ```

  如果你安装了 `pkg-config` ，你可以运行：

  ```
  &lt;samp class="literal"&gt;$&lt;/samp&gt; pkg-config --cflags libpq
  &lt;samp class="literal"&gt;-I/usr/local/include&lt;/samp&gt;
  ```

  注意路径的前面早已包括 `-I` 。

  如果没能给编译器提供正确的选项将产生类似下面这样的错误信息：

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- 在链接最后的程序的时候，声明选项 `-lpq` ，这样就可以把 libpq库链接进来，还要声明 ``` -L``_directory_ ``` 以告诉编译器 libpq所处的目录。(同样，编译器也会搜索一些缺省的目录。) 为了尽可能提高可移植性，你应该把 `-L` 选项放在 `-lpq` 选项前面。比如：

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  你也可以用 `pg_config` 找出库目录：

  ```
  &lt;samp class="literal"&gt;$&lt;/samp&gt; pg_config --libdir
  &lt;samp class="literal"&gt;/usr/local/pgsql/lib&lt;/samp&gt;
  ```

  或再次使用 `pkg-config` ：

  ```
  &lt;samp class="literal"&gt;$&lt;/samp&gt; pkg-config --libs libpq
  &lt;samp class="literal"&gt;-L/usr/local/pgsql/lib -lpq&lt;/samp&gt;
  ```

  请注意，这会输出全部选项，而不只是路径。

  指向这类问题的错误信息会是类似下面这个样子。

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  这意味着你忘记了 `-lpq` 。

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  这意味着你忘记 `-L` 了或没有指定正确的目录。

31.21. 例子程序

这些例子和其他的可以在字典 `src/test/examples` 的源代码分布中找到。

Example 31-1. libpq 例子程序 1

```
/*
 * testlibpq.c
 *
 *      <!-- Test the C version of libpq, the PostgreSQL frontend library. -->测试libpq的C
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     <!--
 * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
 -->
 * 如果用户在命令行上提供了一个参数，则拿它当作 conninfo 字串使用；
 * 否则缺省为 dbname=postgres 并且使用环境变量或者所有其它连接参数
 * 都使用缺省值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* <!-- Make a connection to the database -->连接数据库 */
    conn = PQconnectdb(conninfo);

    /* <!-- Check to see that the backend connection was successfully made -->检查后端连接成
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     <!--
 * Our test case here involves using a cursor, for which we must be inside
     * a transaction block.  We could do the whole thing with a single
     * PQexec() of "select * from pg_database", but that's too trivial to make
     * a good example.
 -->
```

```
* 我们的测试实例涉及游标的使用，这个时候我们必须使用事务块。
* 我们可以把全部事情放在一个  "select * from pg_database"
* PQexec() 里，不过那样太简单了，不是个好例子。
    */

   /* <!-- Start a transaction block -->开始一个事务块 */
   res = PQexec(conn, "BEGIN");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   /*
    <!--
* Should PQclear PGresult whenever it is no longer needed to avoid memory
    * leaks
-->
* 应该在结果不需要的时候 PQclear PGresult，以避免内存泄漏
    */
   PQclear(res);

   /*
    <!-- Fetch rows from pg_database, the system catalog of databases -->
* 从系统表 pg_database（数据库的系统目录）里抓取数据
    */
   res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }
   PQclear(res);

   res = PQexec(conn, "FETCH ALL in myportal");
   if (PQresultStatus(res) != PGRES_TUPLES_OK)
   {
       fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   /* <!-- first, print out the attribute names -->首先，打印属性名称 */
   nFields = PQnfields(res);
   for (i = 0; i < nFields; i++)
       printf("%-15s", PQfname(res, i));
   printf("\n\n");

   /* <!-- next, print out the rows -->然后打印行 */
   for (i = 0; i < PQntuples(res); i++)
   {
       for (j = 0; j < nFields; j++)
           printf("%-15s", PQgetvalue(res, i, j));
       printf("\n");
   }

   PQclear(res);

   /* <!-- close the portal ... we don't bother to check for errors ... -->关闭入口 ... 我
   res = PQexec(conn, "CLOSE myportal");
   PQclear(res);

   /* <!-- end the transaction -->结束事务 */
   res = PQexec(conn, "END");
   PQclear(res);

   /* <!-- close the connection to the database and cleanup -->关闭数据库连接并清理 */
   PQfinish(conn);

   return 0;
```

```
}
```

Example 31-2. libpq 例子程序 2

```
/*
 * testlibpq2.c
 *      <!-- Test of the asynchronous notification interface -->测试异步通知接口
 *
 * <!-- Start this program, then from psql in another window do -->运行此程序，然后从另外一个
 *   NOTIFY TBL2;
 * <!-- Repeat four times to get this program to exit. -->重复四次，直到程序退出
 *
 <!--
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 -->
 * 或者，如果你想好玩一点，用下面命令填充数据库：
 * （在 src/test/examples/testlibpq2.sql 里提供）：
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * <!-- and do this four times: -->然后做四次：
 *
 *   INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     <!--
 * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
 -->
 * 如果用户在命令行上提供了参数，
     * 那么拿它当作 conninfo 字串；否则缺省设置是 dbname=postgres
     * 并且对其它连接使用环境变量或者缺省值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";
```

```
    /* <!-- Make a connection to the database -->和数据库建立连接 */
    conn = PQconnectdb(conninfo);

    /* <!-- Check to see that the backend connection was successfully made -->检查一下与服务
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     <!--
 * Issue LISTEN command to enable notifications from the rule's NOTIFY.
-->
 * 发出 LISTEN 命令打开来自规则 NOTIFY 的通知
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     <!--
 * should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
-->
 * 如果不再需要了，我们应该 PQclear PGresult ，以避免内存泄漏
     */
    PQclear(res);

    /* <!-- Quit after four notifies are received. -->收到四次通知之后退出。 */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*
         <!--
 * Sleep until something happens on the connection.  We use select(2)
         * to wait for input, but you could also use poll() or similar
         * facilities.
-->
 * 睡眠，直到某些事件发生。我们使用 select(2) 等待输入，
         * 但是也可以用 poll() 或者类似的设施。
         */
        int         sock;
        fd_set      input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break;              /* <!-- shouldn't happen -->不应该发生 */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* <!-- Now check for input -->现在检查输入 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
```

```
                PQfreemem(notify);
                nnotifies++;
            }
        }

        fprintf(stderr, "Done.\n");

        /* <!-- close the connection to the database and cleanup -->关闭数据连接并清理 */
        PQfinish(conn);

        return 0;
}
```

Example 31-3. libpq 例子程序 3

```
/*
 * testlibpq3.c
 *      <!-- Test out-of-line parameters and binary I/O. -->测试外联参数和二进制I/O。
 *
<!--
  * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 -->
  * 在运行这个例子之前，用下面的命令填充一个数据库
 * （在 src/test/examples/testlibpq3.sql 里提供）：
 *
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * <!-- The expected output is: -->期望的输出是：
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <libpq-fe.h>

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
<!--
  * This function prints a query result that is a binary-format fetch from
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 -->
 * 这个函数打印查询结果，这些结果是二进制格式，从上面的
 * 注释里面创建的表中抓取出来的。我们把这个函数单独拆出来
```

```
 * 是因为 main() 函数用了它两次。
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* <!-- Use PQfnumber to avoid assumptions about field order in result -->使用 PQfnumb
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* <!-- Get the field values (we ignore possibility they are null!) -->获取字段值（
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         <!--
 * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
 -->
 * INT4 的二进制表现形式是网络字节序，
         * 我们最好转换成本地字节序。
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         <!--
 * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
 -->
 * TEXT 的二进制表现形式是，嗯，文本，因此 libpq 足够给它附加一个字节零，
         * 因此把它看做 C 字串就挺好。
         *
         * BYTEA 的二进制表现形式是一堆字节，里面可能包含嵌入的空值，
         * 因此我们必须注意字段长度。
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
```

```
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     <!--
 * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
-->
 * 如果用户在命令行上提供了参数,
     * 那么拿它当作 conninfo 字串；否则缺省设置是 dbname=postgres
     * 并且对其它连接参数使用环境变量或者缺省值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* <!-- Make a connection to the database -->和数据库建立连接 */
    conn = PQconnectdb(conninfo);

    /* <!-- Check to see that the backend connection was successfully made -->检查一下与服务
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     <!--
 * The point of this program is to illustrate use of PQexecParams() with
     * out-of-line parameters, as well as binary transmission of data.
     *
     * This first example transmits the parameters as text, but receives the
     * results in binary format.  By using out-of-line parameters we can
     * avoid a lot of tedious mucking about with quoting and escaping, even
     * though the data is text.  Notice how we don't have to do anything
     * special with the quote mark in the parameter value.
-->
 * 这个程序是用来演示使用外联参数的 PQexecParams(),
     * 以及数据的二进制传输。第一个例子使用文本传输参数,
     * 但是用二进制格式接收结果。通过使用外联参数，我们可以避免大量
     * 枯燥的字串的引用和逃逸，即使数据是文本。请注意我们这里不需要对参数值里的引号
     * 做任何特殊的处理。
     */

    /* <!-- Here is our out-of-line parameter value  -->这里是我们的外联参数值*/
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1,       /* <!-- one param -->一个参数 */
                       NULL,    /* <!-- let the backend deduce param type -->让后端推出参数
                       paramValues,
                       NULL,    /* <!-- don't need param lengths since text -->因为是文本,
                       NULL,    /* <!-- default to all text params -->缺省是全部文本参数 */
                       1);      /* <!-- ask for binary results -->要求二进制结果 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);
```

```
    PQclear(res);

    /*
     <!--
  * In this second example we transmit an integer parameter in binary
     * form, and again retrieve the results in binary form.
     *
     * Although we tell PQexecParams we are letting the backend deduce
     * parameter type, we really force the decision by casting the parameter
     * symbol in the query text.  This is a good safety measure when sending
     * binary parameters.
  -->
  * 在这个第二个例子里，我们以二进制格式传输一个整数参数，
     * 然后还是以二进制格式检索结果。
     *
     * 尽管我们告诉 PQexecParams，我们让后端推导参数类型，
     * 实际上我们通过在查询字串里转换参数符号的方法强制了决定的做出。
     * 在发送二进制参数的时候，这是一个很好的安全检查。
     */

    /* <!-- Convert integer value "2" to network byte order -->把整数值 "2" 转换成网络字节序
    binaryIntVal = htonl((uint32_t) 2);

    /* <!-- Set up parameter arrays for PQexecParams -->为 PQexecParams 设置参数数组 */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* <!-- binary -->二进制 */

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* <!-- one param -->一个参数 */
                       NULL,    /* <!-- let the backend deduce param type -->让后端推导参数
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* <!-- ask for binary results -->要求二进制结果 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* <!-- close the connection to the database and cleanup -->关闭与数据库的连接并清理 */
    PQfinish(conn);

    return 0;
}
```

Chapter 32. 大对象

Table of Contents

PostgreSQL有一个*large object*(大对象)设施， 它为存储在特殊的大对象结构里的用户数据提供流状的访问方式。 流访问对那些数据值太大，不能一次性操作的数据是很有用的。

本章描述PostgreSQL 大对象数据的实现以及编程和查询语言接口。 我们在本章中使用libpq的C库作为例子， 但是大多数PostgreSQL内置的接口都支持等效的功能。 其它接口可以在内部使用大对象接口以提供对大对象值的一般性支持。 那些内容没有在这里描述。

32.1. 介绍

所有的大对象存储在单独系统表 `pg_largeobject` 中。 每个大对象在系统表 `pg_largeobject_metadata` 中也有记录。 可以使用类似于文件标准操作的读/写API创建，修改，删除大对象。

PostgreSQL也支持 "TOAST"存储机制， 自动存储大于单独数据页的值到每个表的二级存储区。 这样就令大对象接口在一定程度上过时了。大对象接口剩余的一个优点是它允许数据最大有4T， 而TOAST字段只能处理1G。 并且，大对象读取和更新部分可以有效进行，而TOAST字段的大部分操作将 作为一个单元读取或者写整个值。

32.2. 实现特点

大对象实现把大对象分解成"chunks"，然后把块存放在数据库记录里面。 在随机读写时使用一个B-tree索引保证对正确的块（chunk）号的检索。

一个大对象的块存储不必相邻。 例如，如果应用程序打开一个新的大对象， 寻找抵消1000000， 并且写入几个字节，这不会导致配置1000000个字节的存储；只有块覆盖真实写数据字节的范围。 一个读操作将，然而，读出先于最后出现块的任何未分配的位置为零。 这相当于在Unix文件系统文件中"不充分分配"文件的普通操作。

PostgreSQL 9.0中，大对象有一个所有者和一组访问权限， 使用GRANT和REVOKE管理。 `SELECT` 的权限需要读大对象， `UPDATE` 的权限需要写入或彻底删除。 只有大对象的所有者（或数据库超级用户）可以删除，评论或改变一个大对象的所有者。 为了调整以前版本的兼容操作，参阅lo_compat_privileges运行时参数。

32.3. 客户端接口

本节描述PostgreSQL的libpq客户端接口库提供来访问大对象的设施。 PostgreSQL大对象接口是对Unix文件系统的模仿， 有仿真的 `open` ， `read` ， `write` ， `lseek` 等。

使用这些函数的所有大对象操作必须在一个SQL事务块中发生， 由于大对象的文件描述符对事务的整段时间内是唯一有效的。

如果在执行这些函数的任何一个发生错误的时候， 该函数将返回一个不可能的值，通常为0或1。 一个描述该错误的消息存储在连接对象中， 并且可以用 `PQerrorMessage` 检索。

客户端应用程序使用包含 `libpq/libpq-fs.h` 头文件的这些函数，并且 可以与libpq库连接。

32.3.1. 创建大对象

函数

```
Oid lo_creat(PGconn *conn, int mode);
```

创建一个新的大对象。返回值是赋予新大对象的OID， 或者是失败的时候是 `InvalidOid` （零）。在PostgreSQL 8.1里， 没有再使用 `_mode_` ，并且它被忽略； 不过，为了和早期的版本向下兼容， 我们最好将其设置为 `INV_READ` , `INV_WRITE` , 或者 `INV_READ | INV_WRITE` 。 （这些符号常量在头文件 `libpq/libpq-fs.h` 里定义。）

例如：

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

函数

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

也创建一个新的大对象。要赋予数值的OID可以用 `_lobjId_` 声明； 如果这么做，那么在该OID已经被其他大对象使用的情况下就会生成错误。 如果 `_lobjId_` 为 `InvalidOid` （零）， 那么 `lo_create` 赋予一个未用的OID，这个和 `lo_creat` 的行为一致。） 返回值是赋予新的大对象的OID，或者是失败情况下的 `InvalidOid` （零）。

`lo_create` 函数是PostgreSQL 8.1里面新增加的； 如果在老的服务器上运行这个函数，它会失败并返回 `InvalidOid` 。

例子：

```
inv_oid = lo_create(conn, desired_oid);
```

32.3.2. 输入大对象

要把一个操作系统文件输入成为大对象，调用

```
Oid lo_import(PGconn *conn, const char *filename);
```

`_filename_` 参数指明要被输入成为大对象的操作系统文件路径名。 返回值是赋予新大对象的OID。 如果失败则返回 `InvalidOid` （零）。请注意这个文件是由客户端接口库读取的， 而不是服务器端；因此它必须存在于客户端文件系统上并且可以被客户应用读取。

函数

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

也能引入一个新的大对象。要赋予数值的OID可以用 `_lobjId_` 声明； 如果这么做，那么在该OID已经被其他大对象使用的情况下就会生成错误。 如果 `_lobjId_` 为 `InvalidOid` （零）， 那么 `lo_import_with_oid` 赋予一个未用的OID，这个和 `lo_import` 的行为一致。） 返回值是赋予新的大对象的OID，或者是失败情况下的 `InvalidOid` （零）。

`lo_import_with_oid` 是PostgreSQL 8.4里面新增加的， 并且内部调用 `lo_create` (8.1新增的)， 如果此功能在8.0或之前运行这个函数，它会失败并返回 `InvalidOid` 。

32.3.3. 输出大对象

要把一个大对象输出为操作系统文件，调用

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

`lobjId` 参数指明要输出的大对象OID， `filename` 参数指明操作系统文件的路径名。请注意这个文件是由客户端接口库写入的， 而不是服务器端。成功时返回1，失败时返回-1。

32.3.4. 打开一个现有的大对象

要打开一个现存的大对象读写，调用

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

参数 `lobjId` 指明要打开的大对象的OID（对象标识）。 `mode` 位控制该对象是用于只读（ `INV_READ` ）， 只写（ `INV_WRITE` ）还是读写。 （这些符号常量在头文件 `libpq/libpq-fs.h` .里定义。） `lo_open` 返回非负大对象标识用于 以后的 `lo_read` ， `lo_write` ， `lo_lseek` ， `lo_lseek64` ， `lo_tell` ， `lo_tell64` ， `lo_truncate` ， `lo_truncate64` 和 `lo_close` 。 这个描述符只是在当前事务中有效。 失败的时候，返回-1。

服务器目前并不区分 `INV_WRITE` 和 `INV_READ` `|` `INV_WRITE` 模式： 可以从这些任一模式中读取描述符。但与 `INV_READ` 有明显的不同： 对于 `INV_READ` ， 你不能写入描述符， 并且从其中读取的数据将反映执行 `lo_open` 的时候事务快照对应的大对象的数据， 而不会考虑本次事务后面写入的或者其他事务写入的数据。 从一个用 `INV_WRITE` 打开的描述符里面读取的数据反映所有其他 已经提交的事务和当前事务的写操作写入的大对象的数据。 这个行为类似普通SQL语句 `SELECT` 在事务模式中 `REPEATABLE READ` 对比 `READ COMMITTED` 的行为。

例子：

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

32.3.5. 向大对象中写数据

函数

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

从 `buf` 中（这个大小必须是 `len` ） 向大对象描述符 `fd` 写入 `len` 字节。 `fd` 参数必须是前面的一个 `lo_open` 调用的返回。 返回实际写入的字节数（当前实现过程中，它总是等于 `len` ，除非有错误）。 出错时，返回值是-1。

尽管 `len` 参数被声明为 `size_t` ， 这个函数将拒绝长度值大于 `INT_MAX` 。实践中，最好以兆字节传输块中数据。

32.3.6. 从大对象中读取数据

函数

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

从大对象描述符 `fd` 中读取 `len` 字节数据到 `buf` 中(大小必须是 `len`)。 `fd` 参数必须是前面的一个 `lo_open` 调用的返回。 返回实际读取的字节数。如果大对象的结尾达到第一，则小于 `len` 。 出错时，返回值是-1。

尽管 `len` 参数被声明为 `size_t` ，这个函数将拒绝长度值 大于 `INT_MAX` 。实践中，最好以兆字节传输块中数据。

32.3.7. 大对象中查找

要改变与一个大对象描述符相关的读写位置，调用

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

这个过程把当前 `fd` 代表的大对象描述符位置指针移动到 `offset` 指明的新的位置。 参数 `whence` 的合法的取值是 `SEEK_SET` （从对象开头开始找）， `SEEK_CUR` （从当前位置开始找）， 和 `SEEK_END` （从对象结尾开始找）。 返回值是新位置指针，如果出错为-1。

当处理可能超过2GB的大对象时，而是使用：

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

这个函数有 `lo_lseek` 同样的操作，但是它可以接受 `offset` 大于2GB和/或者 传递大于2GB的一个结果。注意如果新的位置指针大于2GB，那么 `lo_lseek` 将失败。

`lo_lseek64` 是PostgreSQL 9.3中新的。如果这个函数 运行在一个旧的服务器版本上，则将失败并且返回-1。

32.3.8. 获取一个大对象的当前索引位置

要获取一个大对象描述符的当前读或写位置，调用

```
int lo_tell(PGconn *conn, int fd);
```

如果有错误，返回值是-1。

当处理可能超过2GB的大对象时，而是使用

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

这个函数有 `lo_tell` 的相同操作。但是它传递大于2GB的结果。 注意如果当前读/写位置大于2GB，则 `lo_tell` 失败。

`lo_tell64` 是PostgreSQL 9.3中新的。如果 这个函数在一个旧服务器版本上运行，它将失败并且返回-1。

32.3.9. 截断一个大对象

截断一个给定长度的大对象，调用

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

截断大对象描述符 `fd` 到长度 `len` 的大对象， `fd` 参数必须通过先前的 `lo_open` 返回。 如果 `len` 大于当前大对象的长度，大对象延长到空字节('\0')。 一旦成功， `lo_truncate` 返回零。错误时，返回值是-1。

与描述符 `fd` 相联系的读/写位置没有被改变。

尽管 `len` 参数被声明为 `size_t` ，则 `lo_truncate` 将拒绝长度值大于 `INT_MAX` 。

当处理可能超过2GB大小的大对象时，而是使用

```
int lo_truncate64(PGcon *conn, int fd, pg_int64 len);
```

这个函数和 `lo_truncate` 有同样操作，但是它可以接受 超过2GB的 `len` 值。

`lo_truncate` 是PostgreSQL 8.3中新的； 如果这个函数在一个旧的服务器版本上运行，它将失败并且返回-1。

`lo_truncate64` 是PostgreSQL 9.3中新的； 如果这个函数在一个旧的服务器版本上运行，它将失败并且返回-1。

32.3.10. 关闭一个大对象描述符

一个大对象描述符关闭可以通过调用

```
int lo_close(PGconn *conn, int fd);
```

`fd` 是通过 `lo_open` 返回的大对象描述符。 成功时， `lo_close` 返回零。失败时，返回值是-1。

任何在事务结尾时仍然打开的大对象描述符将自动关闭。

32.3.11. 删除一个大对象

从数据库中删除一个大对象，调用

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

`lobjId`参数声明要删除的大对象的OID。成功时返回1，失败时返回-1。

32.4. 服务器端函数

还有一些对应上面那些客户端函数的服务器端函数， 可以在SQL命令里使用；实际上， 大多数客户端函数都只是服务器端函数的等效接口。这些服务器端函数中， 通过SQL命令调用的实际有用的是 `lo_creat`，`lo_create`， `lo_unlink`， `lo_import` 和 `lo_export` 。 下面是一些例子：

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);       -- returns OID of new, empty large object

SELECT lo_create(43213);   -- attempts to create large object with OID 43213

SELECT lo_unlink(173454);  -- deletes large object with OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- same as above, but specify OID to use
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

服务器端的 `lo_import` 和 `lo_export` 函数和客户端的那几个有着显著的不同。 这两个函数在服务器的文件系统里读写文件， 使用数据库所有者的权限进行。 因此，只有超级用户才能使用他们。相比之下，客户端的输入和输出函数在客户端的文件系统里读写文件， 使用客户端程序的权限。客户端函数不需要超级用户权限。

`lo_read` 和 `lo_write` 的功能通过服务器端调用可用， 但是服务器端函数名不同于客户端接口，因为他们不包含下划线。你必须作为 `loread` 和 `lowrite` 调用这些函数。

32.5. 例子程序

Example 32-1是一个例子程序，显示如何使用libpq里面的大对象接口。 程序的一部分是注释掉的，但仍然保留在源码里面供读者参考。 这个程序可以在源码发布的 `src/test/examples/testlo.c` 里找到。

Example 32-1. libpq的大对象例子程序

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *--------------------------------------------------------------
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
```

```
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = ' ';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\n",
                lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
```

```
}

/*
 * exportFile
 *    export large object "lobjOid" to file "out_filename"
 *
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the large object
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\n",
                lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file %s\n",
                filename);
    }

    /*
     * read in from the inversion file and write to the Unix file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;
```

```
    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 33. ECPG - 在C中嵌入SQL

Table of Contents

这章描写一种用于PostgreSQL的嵌入SQL包。 它是由Linus Tolke(<linus@epact.se>)和 Michael Meskes(<meskes@postgresql.org>)写的。 最初它是为了在C里面使用书写的。它也可以用于C++， 但是它还不能识别所有C++构造。

这份文档相当不完整。但是因为这个接口是标准， 所以我们可以在有关SQL的资源里找到许多额外的信息。

33.1. 概念

嵌入SQL程序主要由一种普通的编程语言代码组成， 在我们这个场合中是C，并且在其中与一些特殊标记的段混合。 要制作这样的程序，源代码(`*.pgc`) 首先经过嵌入的SQL预处理器处理， 它把源代码转换成普通的C程序(`*.c`)， 然后这个程序可以用C编译器进行处理。 关于编译和链接的细节参阅Section 33.10。 转变的ECPG应用在libpq库中通过嵌入的SQL库（ecpglib）调用函数。 并且使用正常的前后端协议与PostgreSQL服务器通信。

嵌入的SQL相比于其它的从 C代码中处理SQL命令的优点有几条。 首先，它照看那些从你的C 程序中的变量中传来传去数值的事情。 第二，在编译时检查程序中SQL代码句法正确性。 第三，在C代码里嵌入SQL是定义 在SQL标准里的， 并且被许多其它的SQL数据库支持。 PostgreSQL的实现被设计成尽可能匹配这个标准， 并且通常可以把为其它SQL数据库书写的SQL移植到 PostgreSQL中来，反之亦然。

如上所述，为嵌入SQL 接口写的程序通常是带着插入进来的特殊代码的C程序， 这些特殊代码用于执行与数据库相关的动作。 这些特殊代码通常的形式是下面这样：

```
EXEC SQL ...;
```

这些语句语法上占据C语句的位置。根据具体语句的不同， 它们可以出现在全局环境中或者出现在一个函数里。 嵌入的SQL语句遵循普通SQL 代码的大小写敏感规则，而不是遵循C代码的。

下面的小节都是用来解释所有的嵌入SQL语句的。

33.2. 管理数据库连接

本节描述了如何打开，关闭，以及切换数据库连接。

33.2.1. 与数据库服务器连接

用下面的语句与一个数据库连接：

```
EXEC SQL CONNECT TO _target_ [AS `_connection-name_`] [USER `_user-name_`];
```

`_target_` 可以通过下面的方法声明：

- `_dbname_` [@ `_hostname_`][: `_port_`]
- `tcp:postgresql://``_hostname_` [: `_port_` ][/ `_dbname_` ][? `_options_`]
- `unix:postgresql://``_hostname_` [: `_port_` ][/ `_dbname_` ][? `_options_`]
- 一个包含上面形式的SQL字串文本
- 一个对包含上面的形式之一的字符串变量的引用
- `DEFAULT`

如果你用文本声明连接目标（也就是说，不是通过一个变量引用）， 而且你也不引用这个数值，那么使用普通SQL的大小写无关的规则。 这种情况下，你也可以根据需要独立地对参数使用双引号包围。 实际上，可能用一个（单引号包围）的 字串文本或者变量引用作为连接目标可能更结实一些。 连接目标 `DEFAULT` 发起一个用缺省用户名对缺省数据库地连接。 这个时候不应该声明用户名或连接名。

声明用户名的方法也有几种不同方式：

- `_username_`
- `_username_` / `_password_`
- `_username_` IDENTIFIED BY `_password_`
- `_username_` USING `_password_`

正如上面的一样，参数用户名和密码可以是一个SQL标识， 一个字符变量，或者一个字符串。

`_连接名_` 用于处理一个程序里的多个连接。 如果一个程序只使用一个连接，则可以省略它。最近打开的连接成为当前连接， 在准备执行SQL语句的时候，缺省时会使用这个连接（参阅本章稍后部分）。

这里是一些 `CONNECT` 语句的例子：

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* or EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最后的一个形式使用了上面说过的变量引用的方法。 在后面的小节里你会看到在SQL语句里如何使用前缀了冒号的C变量。

请注意连接目标的格式没有在SQL标准里说明。所以， 如果你想开发可以移植的应用， 你可能会想使用类似上面的最后一个例子这样的方法来把连接目标字串封装在某处。

33.2.2. 选择一个连接

在当前连接中缺省执行嵌入SQL程序的SQL语句，也就是说，最近打开的。 如果应用需要管理多个连接，那么有两种处理方法。

第一个选项是为每个SQL语句明确选择一个连接，比如:

```
EXEC SQL AT _connection-name_ SELECT ...;
```

如果在混合顺序中应用程序需要使用若干个连接时，这个选项特别适合。

如果你的应用程序使用多个执行线程，他们不能同时共享连接。 你要么明确控制访问连接（使用互斥锁）或者为每个线程使用一个连接。 如果每个线程使用自己的连接，你将需要使用AT子句指定线程将使用哪个连接。

第二个选项是执行语句切换当前连接。语句是：

```
EXEC SQL SET CONNECTION _connection-name_;
```

如果在同一个连接上执行许多语句，那么这种选择很方便。它不是线程感知的。

这里有一个管理多个数据库连接的例子程序：

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;

/*在最后打开的数据库"testdb3"中执行该查询*/
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

/*在"testdb2"中使用"AT"运行查询*/

    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

/*切换当前连接到"testdb1" */

    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子可能产生这样的输出:

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

33.2.3. 关闭一个连接

使用下面的语句关闭连接：

```
EXEC SQL DISCONNECT [`_connection_`];
```

通过下面的方法声明 `_连接_`：

- `_连接名_`
- `缺省`
- `当前`
- `所有`

如果没有声明连接名，那么关闭当前连接。

应用保持明确关闭每次打开的连接是一个很好的习惯。

33.3. 运行SQL命令

在嵌入的SQL应用中可以运行任何SQL命令。下面是一些如何使用它们的例子。

33.3.1. 执行SQL语句

创建一个表：

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

插入一行：

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

删除一行：

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

更新：

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

可以通过`EXEC SQL`直接执行返回一个结果行的`SELECT`语句。为了处理多行结果集，应用程序必须使用游标；参阅Section 33.3.2。（特殊情况下，应用程序可以一次读取多行到数组宿主变量中；参阅Section 33.4.4.3.1。）

单行select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

同时，可以使用`SHOW`命令检索配置参数：

```
EXEC SQL SHOW search_path INTO :var;
```

形如 `:``_something_` 这样的记号是宿主变量，也就是说，他们指向C程序中的变量。在Section 33.4中有解释。

33.3.2. 使用游标

为了检索出多行的结果集，应用程序必须声明一个游标并且从游标中抓取每一行。 使用游标的步骤如下：声明一个游标，打开它，从游标中抓取一行，重复，最后关闭它。

使用游标选择：

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

关于游标声明的更多细节，请参阅DECLARE，关于 `FETCH` 命令的细节请参阅FETCH。

> **Note:** ECPG `DECLARE` 命令实际上不会造成语句被发送到PostgreSQL后端。 当执行 `OPEN` 命令时， 在后端(使用后端的 `DECLARE` 命令)打开游标。

33.3.3. 管理事务

在缺省模式下，语句只有在 `EXEC SQL COMMIT` 发出的时候才提交， 嵌入的SQL接口也支持事务的自动提交（类似libpq的行为）， 方法是通过给 `ecpg` （见ecpg） 增加命令行选项 `-t` ，或者是通过 `EXEC SQL SET AUTOCOMMIT TO ON` 语句。 在自动提交模式里，每条命令都是自动提交的， 除非它们包围在一个明确的事务块里。 这个模式可以用 `EXEC SQL SET AUTOCOMMIT TO OFF` 明确地关闭。

有以下事务管理命令可用：

`EXEC SQL COMMIT`

提交正在进行的事务。

`EXEC SQL ROLLBACK`

回滚正在进行的事务。

`EXEC SQL SET AUTOCOMMIT TO ON`

启动自动提交模式。

`SET AUTOCOMMIT TO OFF`

禁用自动提交模式。这是缺省的。

33.3.4. 预备语句

当编译时间不知道该值已被传递给SQL语句，或者同一语句将使用多次， 那么预备语句是有帮助的。

使用命令 `PREPARE` 准备语句。对于不知道的值， 使用占位符" `?` ":

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

如果一个语句返回单行，应用程序可以在 `PREPARE` 执行语句之后调用 `EXECUTE` ， 使用 `USING` 子句为占位符提供实际值：

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

如果一个语句返回多行，应用程序可以使用基于预备语句声明的游标。 为了结合输入参数，必须使用 `USING` 子句打开游标：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* 当结果集达到最后时，打破while循环 */

EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

当你不再需要预备语句的时候，你应该重新分配它：

```
EXEC SQL DEALLOCATE PREPARE _name_;
```

为获得关于 `PREPARE` 的更多详情， 请参阅PREPARE。同时参阅Section 33.5 获得关于占位符和输入参数的更多详情。

33.4. 使用宿主变量

在Section 33.3 里你看到了如何从嵌入的SQL程序里执行SQL语句。 那些语句有些只使用了固定的数值， 并没有提供一个插入用户提供的数值到语句中的方法， 也没有提供让程序访问查询返回的数值的方法。 这种类型的语句在实际应用中并不是很有用。 本节详细解释如何在你的C程序和嵌入的SQL语句之间使用 一种被称作宿主变量的机制传递数据。在嵌入SQL程序中， 我们将SQL语句认为是宿主语言_C程序编码的客人。 因此C程序变量称为宿主变量_。

在PostgreSQL后端和ECPG应用程序之间改变值的另一种方式是使用SQL描述符， 参见Section 33.7中的描述。

33.4.1. 概述

在C程序和SQL语句之间传递数据在嵌入的 SQL 里是特别简单的。 我们不用把数据粘贴到语句中，这样必然会有各种复杂事情需要处理， 比如正确地给数值加引号等等，我们只需要在SQL语句里写上C变量的名字， 前缀一个冒号即可。比如：

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

这个语句引用了两个变量,一个叫 `v1`， 另一个叫 `v2`， 并且也使用一个普通的SQL字串文本，这样表明你并不局限于只使用某一种数据或者其他。

这种在SQL语句里插入C变量的方式在SQL语句里任何需要表达式的地方都可用。

33.4.2. 声明段

要从程序向数据库传递数据，比如，查询中的参数， 或者从数据库里向程序传回的数据，想包含这类数据的 C变量必须在一个特殊的标记段里面声明， 这样嵌入的SQL预处理器就会明白要做什么。

这个段以下面的代码开头：

```
EXEC SQL BEGIN DECLARE SECTION;
```

以下面的代码结束:

```
EXEC SQL END DECLARE SECTION;
```

在这些行之间，有普通的C变量声明，比如：

```
int   x = 4;
char  foo[16], bar[16];
```

正如你所看到的，你可以随意指定一个初始值给变量。 变量的范围是在程序中通过其声明部分的位置确定。 你也可以用下面的语法，隐式地创建一个声明段声明变量：

```
EXEC SQL int i = 4;
```

在程序里你可以有任意多个声明段。

这些声明也同时以普通C变量的形式回显到输出文件中， 因此，我们不必再声明他们。 那些不准备在SQL命令里使用的变量通常可以在这些特殊的段外面声明。

结构或者联合的定义也必须在 `DECLARE` 段中列出。 否则，预处理器就无法处理这些类型，因为它不知道定义。

33.4.3. 检索查询结果

现在你应该能把你的程序生成的数据传递到SQL命令里面去了。 但是你如何检索一个查询的结果呢？为了这个目的， 嵌入的SQL提供了常用命令 `SELECT` 和 `FETCH` 的特殊变体。 这些命令有了特殊的 `INTO` 子句， 声明检索出来的数值存储在哪个宿主变量里。 `SELECT` 用于返回单行的查询，同时 `FETCH` 用于使用游标返回多行的查询。

下面是一个例子：

```
/*
 * 假设表是这个:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

所以 `INTO` 子句出现在选择列表和 `FROM` 子句之间。 选择列表和 `INTO` 后面的列表的元素 （也叫目标列表）个数必须相同。

下面是使用 `FETCH` 命令的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

这里的 `INTO` 子句出现在所有正常的子句后面。

33.4.4. 类型映射

当ECPG应用程序改变PostgreSQL服务器和C应用程序之间的值的时候， 比如检索来自服务器的查询结果或者执行带有输入参数的SQL语句， 在PostgreSQL数据类型和宿主语言变量类型（具体地C语言数据 类型）之间需要改变值。ECPG的一个主要点之一是 在大多数情况下自动的关注这个。

在这方面，有两种数据类型：一些简单的 PostgreSQL数据类型，如 `integer` 和 `text` ， 可以直接通过应用程序读取和写入。 其他PostgreSQL数据类型，如 `timestamp` 和 `numeric` 只能通过特殊库函数进行访问；参阅Section 33.4.4.2。

Table 33-1显示了哪个PostgreSQL 数据类型对应哪个C数据类型。当你希望 发送或接收一个给定PostgreSQL数据类型的值时，你应该 在声明部分声明一个对应C数据类型的C变量。

Table 33-1. PostgreSQL数据类型和C变量类型之间的映射

PostgreSQL数据类型	宿主变量类型
`smallint`	`short`
`integer`	`int`
`bigint`	`long long int`
`decimal`	`decimal` [a]
`numeric`	`numeric` [a]
`real`	`float`
`double precision`	`double`
`smallserial`	`short`
`serial`	`int`
`bigserial`	`long long int`
`oid`	`unsigned int`
`character(``_n_ )`, `varchar(``_n_)`, `text`	`char[``_n_ +1]`, `VARCHAR[``_n_ +1]`[b]
`name`	`char[NAMEDATALEN]`
`timestamp`	`timestamp` [a]
`interval`	`interval` [a]
`date`	`date` [a]
`boolean`	`bool` [c]

Notes: a. 这种类型可以通过特殊库函数访问；参阅Section 33.4.4.2。 b. 在 `ecpglib.h` 中声明 c. 如果不是本地的，在 `ecpglib.h` 中声明

33.4.4.1. 处理字符串

为了处理SQL字符串数据类型，比如 `varchar` 和 `text` ， 有两种可能方式声明宿主变量。

一种方式是使用 `char[]` ， `char` 数组是在C中处理字符数据 最常见方式。

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

请注意，你必须关注自身长度。 如果你使用这个宿主变量 作为查询返回一个具有多于49个字符的字符串的目标变量， 那么发生缓冲区溢出。

另一种方法是使用 `VARCHAR` 类型，这是一个由ECPG提供的 特殊类型。 `VARCHAR` 类型的数组定义被转换为 每个变量的命名 `结构` 。声明如：

```
VARCHAR var[180];
```

转换成：

```
struct varchar_var { int len; char arr[180]; } var;
```

`arr` 有一个终止零字节的字符串。 因此，为了在 `VARCHAR` 宿主变量中存储字符串， 宿主变量必须声明为包含零字节终结符的长度。 `len` 持有 存储在 `arr` 中而没有 终止零字节的字符串长度。当一个宿主变量作为一个查询输入时， 如果 `strlen(arr)` 和 `len` 是不同的，那么使用稍短的。

两个或以上 `VARCHAR` 宿主变量不能在单行语句中被声明。 下面的代码将混淆 `ecpg` 预处理程序：

```
VARCHAR v1[128], v2[128];   /* WRONG */
```

两个变量应该像下面这样在独立语句中进行定义：

```
VARCHAR v1[128];
VARCHAR v2[128];
```

`VARCHAR` 可以使用大写或小写，但是在不混淆的情况下。

`char` 和 `VARCHAR` 宿主变量可以持有其它SQL类型的值， 这将被存储在它们的字符串形式中。

33.4.4.2. 访问特定数据类型

ECPG含有一些特定类型帮助你 与来自PostgreSQL服务器的一些特殊数据类型进行轻松互动。 特别是，它已经实现支持 `numeric` , `decimal` , `date` , `timestamp` 和 `interval` 类型。 这些数据类型不能有效地映射到原始主机变量类型（例如 `int` , `long long int` 或者 `char[]` ）， 因为他们有一个复杂的内部结构。 应用程序通过声明特殊类型的主机变量处理这些类型，并且在pgtypes库中使用函数访问他们。 该pgtypes库包含处理这些类型的基本函数的 详细描述参阅Section 33.6， 这样你就不需要发送一个查询到SQL服务器，仅仅为了添加间隔时间戳例子。

以下小节描述了这些特殊数据类型。为了获得关于pgtypes库函数的更多细节， 参阅Section 33.6。

33.4.4.2.1. timestamp, date

这是在ECPG宿主应用程序中处理 `timestamp` 变量的模式。

首先，程序必须包含 `timestamp` 类型的头文件：

```
#include <pgtypes_timestamp.h>
```

接下来，在声明部分声明作为类型 `timestamp` 的宿主变量：

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

并且读取值到宿主变量之后， 使用pgtypes库函数处理它。在下面的例子中， 使用 `PGTYPEStimestamp_to_asc()` 函数该 `timestamp` 值转换成文本（ASCII）形式：

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

这个例子将显示如下一些结果：

```
ts = 2010-06-27 18:03:56.949343
```

此外，日期类型可以用同样的方式处理。 程序必须包括 `pgtypes_date.h`， 作为日期类型声明一个宿主变量并且使用 `PGTYPESdate_to_asc()` 函数转换日期值为文本形式。 关于pgtypes库函数的更多详情，请参阅Section 33.6。

33.4.4.2.2. interval

`interval` 类型 的处理也与 `timestamp` 和 `date` 类型类似。然而，为了 `interval` 类型值显式分配内存是必需的。换句话说， 该变量的存储空间在堆内存中被分配，而不是在堆栈存储器中。

下面是一个示例程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.2.3. numeric, decimal

`numeric` 和 `decimal` 类型的处理类似于 `interval` 类型：它需要定义一个指针， 在堆上分配一些内存空间，并且使用pgtypes库函数访问 变量。关于pgtypes库函数的更多细节，参阅Section 33.6。

对于 `decimal` 类型没有提供专门的函数。 应用程序使用pgtypes库函数做进一步的处理 将其转换成 `numeric` 变量。

这里有一个处理 `numeric` 和 `decimal` 类型变量的示例程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

/*转换十进制到数值型以显示十进制值*/
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.3. 使用非初级类型的宿主变量

作为一个宿主变量你也可以使用数组，typedefs，结构和指针。

33.4.4.3.1. Arrays

有两个作为宿主变量的数组用例。 最先的一种方式是在 `char[]` 或者 `VARCHAR[]` 中存储一些文本字符串， 正如Section 33.4.4.1解释的。 第二个用例是不使用游标从查询结果检索多行。 没有一个数组处理包括多行的一个查询结果， 它需要使用一个游标和 `FETCH` 命令。 但使用数组宿主变量，一次可以检索多行。 数组长度被定义为能够容纳所有行，否则可能会发生缓冲区溢出。

下面的示例扫描 `pg_database` 系统表并且显示所有OID和可用数据库的名字：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;

/*同时检索多行到数组中*/
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

这个例子显示了如下结果。（精确值取决于区域环境）

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

33.4.4.3.2. 结构

一个成员名称匹配查询结果列名称的结构， 可用于一次检索多个列。该结构可以在单一的宿主变量中处理多个列的值。

下面的示例检索OID，名称，和 来自 `pg_database` 系统表可用数据库的大小，并且使用 `pg_database_size()` 函数。在这个例子中， 一个结构变量 `dbinfo_t` 和 名称匹配 `SELECT` 结果的每一列的成员是用来检索一个 结果行，而没有把多个宿主变量放在 `FETCH` 声明中。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size F
    EXEC SQL OPEN cur1;

/*当结果集到达末尾时，打破while循环*/
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {

/*抓取多列到一个结构中*/
        EXEC SQL FETCH FROM cur1 INTO :dbval;

/*打印结构成员*/
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

这个例子显示了下面结果。（精确值取决于区域环境）

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

结构宿主变量"合并"和结构一样的许多列 作为结构域。附加的列可以被分配 给其他宿主变量。例如，上述程序可能 也会像这样被重组，使用外部结构 `size` 变量。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size F
    EXEC SQL OPEN cur1;

    /*当结果集到达末尾时，打破while循环*/
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {

/*抓取多列到一个结构中*/
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

/*打印结构成员*/
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

33.4.4.3.3. Typedefs

使用 `typedef` 关键字映射新类型到已有类型。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

注意，你也可以使用：

```
EXEC SQL TYPE serial_t IS long;
```

这种声明并不需要声明部分。

33.4.4.3.4. 指针

你可以声明最常见类型的指针。 然而注意你不能作为没有自动分配的查询目标变量 而使用指针。参阅Section 33.7获取更多自动配置的信息。

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

33.4.5. 处理非初级的SQL数据类型

本节包含了如何处理nonscalar和ECPG应用程序中用户自定义的SQL级别数据类型的相关信息。 请注意这不同于非初级类型宿主变量的处理，在前面的章节中有描述。

33.4.5.1. 数组

在ECPG中不直接支持SQL级别数组。不可能简单的映射SQL数组到C数组宿主变量。 这将产生未定义操作。然而，存在一些解决方法。

如果查询分别访问数组元素，那么这可以避免在ECPG中使用数组。 然后，应该使用可以映射到元素类型的宿主变量。比如， 如果列类型是 `integer` 数组，那么使用 `int` 类型宿主变量。 如果元素类型是 `varchar` 或者 `text`， 则可以使用 `char[]` 或者 `VARCHAR[]` 类型宿主变量。

下面是一个例子。假设下列表:

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

下面示例程序检索了数组的第四个元素，并且将它存储在 `int` 类型 的宿主变量中：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

该例子显示了下面结果：

```
ii=4
```

为了映射多个数组元素到数组列的数组类型宿主变量每个元素的多个元组，并且 宿主变量数组的每个元素必须分别被管理，比如：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

请再次注意

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* WRONG */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

不会在这种情况下正确工作，因为你不能映射一个数组类型列直接到数组宿主变量。

另外一种方法是在 `char[]` 或者 `VARCHAR[]` 类型宿主变量的外部字符串形式中存储数组。 更多关于该形式的详细信息，请参阅Section 8.15.2。 注意这意味着在主程序（没有对分析文本表示的进一步处理）中数组自然不能作为数组被访问。

33.4.5.2. 复合类型

在ECPG中不直接支持复合类型，但是简单解决方法是可能的。 可用的方法与上面数组描述的那个是类似的： 要么分别访问每个属性，要么使用外部字符串表示形式。

下面列子中，假设下面类型和表：

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

最明显的解决方法是分别访问每个属性。下面程序通过分别选择类型 `comp_t` 的每个属性的示例表中检索数据：

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/*将复合类型列的每个元素放在SELECT列表中*/

EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
/*抓取复合类型列的每个元素给宿主变量*/
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

为了加强这个例子，在 `FETCH` 命令中存储值的宿主变量 可以聚集在一个结构中。 关于结构形式中宿主变量的更多细节，参阅Section 33.4.4.3.2。 为了切换到结构，例子可以做如下修改。 两个宿主变量 `intval` 和 `textval` ， 是 `comp_t` 结构成员，并且在 `FETCH` 命令上 指定结构。

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/*将复合类型列的每个元素放在SELECT列表中*/

EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
/*将SELECT列表中所有值放到结构中*/

    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

虽然结构用于 `FETCH` 命令，逐一指定 `SELECT` 子句中的属性名，这可以通过使用 `*` 请求复合类型值的所有属性获得提高。

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
/*将SELECT列表中的所有值放到结构中*/
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

这种方式，可以无缝的将复合类型映射到结构中，尽管ECPG并不了解该复合类型。

最后，在类型 `char[]` 或者 `VARCHAR[]` 宿主变量中的外部字符串表示形式中 存储复合类型是可能的。但是那种方式，不容易从主程序中访问值的字段。

33.4.5.3. 用户自定义基本类型

ECPG不直接支持新用户自定义基础类型。你可以使用外部字符串表示形式和 类型 `char[]` 或者 `VARCHAR[]` 的宿主变量，并且该方法 对于许多类型的确是合适的并且充分的。

这是一个使用Section 35.11中 `复合` 数据类型的例子。 该类型的外部字符串表示形式是 `(%lf,%lf)` ， 定义在Section 35.11中的 `complex_in()` 和 `complex_out()` 函数中。 下面例子将复合类型值 `(1,1)` 和 `(3,3)` 插入到列 `a` 和 `b` 中， 并且之后从表中选择它们。

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

这个例子显示了如下结果:

```
a=(1,1), b=(3,3)
```

另一种方法是避免ECPG中用户自定义类型的直接使用，并且创建一个函数或者计算在 用户自定义类型和ECPG处理的原始类型之间的转换。 注意，然而那个类型计算，特别是隐式的那个，应该小心引入类型系统中。

比如，

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0')' + $2 * complex '(0,1)' $$;
```

这个定义之后，下面

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b), create_complex(:c, :d))
```

具有相同效果正如

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

33.4.6. 指示器

上面的例子不能处理空值。实际上，如果从数据库中抓到一条空值， 那么上面的检索例子会抛出一个错误。 要能够向数据库中传递空值，或者从数据库中检索空值， 你需要给每个包含数据的宿主变量后面附加一个额外的宿主变量。 这第二个宿主变量叫指示器，里面包含一个标志， 告诉我们数据是否为空，如果为空，那么真正的宿主变量的数值就可以忽略。 下面是一个能正确检索空值的例子:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

如果数值不是空，那么指示器变量 `val_ind` 将是零， 如果值是空，那么它将是负数。

指示器还有另外的一个用途，如果指示器值是正数， 则意味着值不空，但是在数值存储到宿主变量里的时候被截断了。

如果参数`-r no_indicator`被传递给预处理器`ecpg`， 那么它在"no-indicator"模式下工作。在非指示器模式下， 如果没有声明可用指示器，那么为了将字符串类型作为空字符串以及 整数类型作为类型的最小可能值（比如， `int`最小为`INT_MIN`），则使用空值（在输入和输出上）。

33.5. 动态SQL

在许多情况下，应用要执行的具体的SQL语句在书写应用的时候就已经知道了。 不过，在某些情况下，SQL语句是在运行时或者由外部的数据提供的。 在这种情况下，我们不能直接在C代码嵌入SQL语句， 但是有个机制可以允许你调用放在一个字串变量里的任何SQL语句。

33.5.1. 执行没有结果集的语句

执行任意SQL语句最简单的方法是使用`EXECUTE IMMEDIATE`命令。 比如:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE`可以用于不返回结果集 （比如，DDL，`INSERT`，`UPDATE`，`DELETE`）的SQL语句。 你不能用这种方式执行检索数据（比如`SELECT`）的语句。 下一节将描述该如何做。

33.5.2. 执行具有输入参数的语句

执行任意SQL语句的更强大的方法是准备这些语句一次， 并且执行这些准备好的语句任意多次。 我们也可以准备一个普遍的语句版本，然后通过替换一些参数， 执行一个特定的版本。 在准备语句的时候， 在你稍后需要替换参数的地方书写一个问号。比如：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

当你不再需要预备语句时，你应该释放它：

```
EXEC SQL DEALLOCATE PREPARE _name_;
```

33.5.3. 执行带有结果集的语句

为了执行具有单独结果集的SQL语句，可以使用`EXECUTE`。 为了保存结果，增加`INTO`子句。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

一个 `EXECUTE` 命令可以有一个 `INTO` 子句， 一个 `USING` 子句，也可以两个都有或者两个都没有。

如果查询希望返回多个结果行，那么使用游标，正如下面例子。 参阅Section 33.3.2获取更多关于游标的信息。

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

33.6. pgtypes 库

Pgtypes库映射PostgreSQL数据库类型到C等值， 它可用于C程序。没有PostgreSQL服务器的帮助下， 它也提供一些函数在C中用这些类型做基本的运算。 请看下面的例子：

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
free(out);
```

33.6.1. 数值类型

数值类型提供任意精度的计算。 参见Section 8.1 获取PostgreSQL服务器等价类型。 由于任意精度这一变量需要能够扩展和动态收缩。 那就是你只能在堆上创建数值变量的原因， 通过 `PGTYPESnumeric_new` 和 `PGTYPESnumeric_free` 函数。 和十进制类型类似但精确度有限，可以在栈中创建也可以在堆上创建。

下列函数用于处理数值类型：

`PGTYPESnumeric_new`

请求一个新分配的数值变量的指针。

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

任意数值类型释放所有内存。

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

从字符串标号解析数值类型。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

有效格式比如： `-2` ， `.794` ， `+3.44` ， `592.49E07` 或者 `-32.84e-4` 。 如果值解析不成功，返回一个有效指针，否则空指针。 此刻ECPG总是解析完整的字符串， 所以目前不支持存储在 `*endptr` 第一无效字符的地址。 你可以安全地设置 `endptr` 为空。

`PGTYPESnumeric_to_asc`

返回一个指向字符串的指针， 该字符串是通过 `malloc` 包 含数值类型 `num` 的字符串表示形式分配的。

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

如果必要的话，数值类型的值将带有 `dscale` 小数位数舍入。

`PGTYPESnumeric_add`

添加两个数值变量到三分之一。

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

函数添加变量 `var1` 和 `var2` 到结果变量 `result` 中。 函数成功时返回0，错误情况下返回-1。

`PGTYPESnumeric_sub`

减去两个数值变量并且返回三分之一结果。

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

函数从变量 `var1` 中减去变量 `var2` 。 操作的结果被存储在变量 `result` 中。 函数成功时返回0，并且错误的情况下返回-1。

`PGTYPESnumeric_mul`

两个数值变量相乘，并且返回三分之一结果。

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

函数将变量 `var1` 和 `var2` 相乘。操作的结果被存储在变量 `result` 中。 函数成功时返回0，并且错误的情况下返回-1。

`PGTYPESnumeric_div`

两个数值变量相除并且返回三分之一结果。

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

函数将变量 `var1` 除以变量 `var2` 。 操作的结果被存储在变量 `result` 中。 函数成功时返回0并且错误的情况下返回-1。

`PGTYPESnumeric_cmp`

比较两个数值变量。

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

这个函数比较两个数值变量。在错误的情况下，返回 `INT_MAX` 。 成功，函数返回三个可能结果之一：

- 如果 `var1` 大于 `var2` ,则返回1。
- 如果 `var1` 小于 `var2` ,则返回-1。
- 如果 `var1` 等于 `var2` ,则返回0。

`PGTYPESnumeric_from_int`

转换一个int变量到数值变量。

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

这个函数接受有符号整型变量并将其存储在数值变量 `var` 中， 成功时，则返回0。在失败的情况下，返回-1。

`PGTYPESnumeric_from_long`

转换长整型变量到数值变量。

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

这个函数接受有符号长整型变量并将其存储在数值变量 `var` 中，成功时， 则返回0。在失败的情况下，返回-1。

`PGTYPESnumeric_copy`

拷贝一个数值变量为另一个变量。

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

这个函数拷贝变量的值，这个变量是 `src` 指向 `dst` 指向的变量， 成功时返回0，错误的情况下返回-1。

`PGTYPESnumeric_from_double`

将double类型的变量转换成数值类型的。

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

这个函数接受double变量并将其结果存储在 `dst` 指向的变量中， 成功时，则返回0。在失败的情况下，返回-1。

`PGTYPESnumeric_to_double`

将数值类型变量转换成double类型的。

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

这个函数从变量中转换数值类型的值， 这个变量是 `nv` 指向的 `dp` 指向的double变量， 成功时返回0，错误的情况下返回-1，包括溢出。溢出的时候， 全局变量 `errno` 将额外设置 `PGTYPES_NUM_OVERFLOW` 。

`PGTYPESnumeric_to_int`

将数值类型变量转化成整型。

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

这个函数从变量中转换数值类型的值，这个变量是 `nv` 指向的 `ip` 指向的整型变量， 成功时返回0，错误的情况下返回-1，包括溢出。溢出的时候， 全局变量 `errno` 将额外设置 `PGTYPES_NUM_OVERFLOW` 。

`PGTYPESnumeric_to_long`

将数值类型的变量转换成long类型。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

这个函数从变量中转换数值类型的值，这个变量是 `nv` 指向的 `lp` 指向的长整型变量， 成功时返回0，错误的情况下返回-1，包括溢出。溢出的时候， 全局变量 `errno` 将额外设置 `PGTYPES_NUM_OVERFLOW` 。

`PGTYPESnumeric_to_decimal`

将数值类型的变量转换成十进制类型。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

这个函数从变量中转换数值类型的值，这个变量是 `src` 指向的 `dst` 指向的十进制变量， 成功时返回0，错误的情况下返回-1，包括溢出。溢出的时候， 全局变量 `errno` 将额外设置 `PGTYPES_NUM_OVERFLOW` 。

`PGTYPESnumeric_from_decimal`

将十进制类型的变量转换成数值类型。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

这个函数从变量中转换十进制值，这个变量是 `src` 指向的 `dst` 指向的数值变量， 成功时返回0，错误的情况下返回-1，包括溢出。 由于十进制类型作为数值类型的有限版本实现的，不会发生这种转换溢出。

33.6.2. 日期类型

C中的日期类型允许你的程序处理SQL类型的数据。参见Section 8.5 获得PostgreSQL服务器的等价类型。

下面的函数可以适用于日期类型：

`PGTYPESdate_from_timestamp`

从一个时间戳中提取日期部分。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

这个函数接受一个时间戳作为其唯一的参数,并且从时间戳返回提取日期部分。

`PGTYPESdate_from_asc`

从文本表示解析日期。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

函数接收C char字符串 `str` 和指向*C char*字符串 `endptr` 的指针。 此刻ECPG总是解析完整的字符串， 所以目前不支持存储在 `*endptr` 中的第一无效字符的地址。 你可以安全地设置 `endptr` 无效。

注意，函数始终假定MDY格式化日期目前在ECPG还没有改变。

Table 33-2显示了允许输入格式。

Table 33-2. `PGTYPESdate_from_asc` 有效输入格式

输入	结果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

`PGTYPESdate_to_asc`

返回一个数据变量的文本表示。

```
char *PGTYPESdate_to_asc(date dDate);
```

函数接收日期 `dDate` 作为它唯一参数。 输出数据的形式 `1999-01-18`， 即 `YYYY-MM-DD` 格式。

`PGTYPESdate_julmdy`

从一个日期型的变量中提取一天、本月和一年的值。

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

函数接收日期 `d` 和一个指向3个整型值 `mdy` 数组的指针。 变量名称显示顺序： `mdy[0]` 设置为包含的几个月份， `mdy[1]` 设置为一天的值， `mdy[2]` 包含一年的值。

`PGTYPESdate_mdyjul`

从指定日期的年、月、日的3个整型值数组中创建一个日期值。

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

函数接收3个整型(`mdy`)的数组作为第一个参数， 第二个参数是指向保留运算结果的日期型变量的指针。

`PGTYPESdate_dayofweek`

返回表示日期值的一个星期数。

```
int PGTYPESdate_dayofweek(date d);
```

函数接收日期变量 `d` 作为其唯一的参数， 并返回一个整数，表示这个日期的本周的一天。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三
- 4 - 星期四
- 5 – 星期五
- 6 – 星期六

`PGTYPESdate_today`

得到当前日期。

```
void PGTYPESdate_today(date *d);
```

函数接收指向日期变量(`d`)的一个指针，它设置当前的日期。

`PGTYPESdate_fmt_asc`

将日期类型变量转换成使用格式掩码的文本表示形式。

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

函数接收一个转换(`dDate`)日期， 格式掩码(`fmtstring`)以及保持日期(`outbuf`)文本表示形式的字符串。

成功的时候返回0，如果产生错误则返回负数。

下列是你可以使用的字段分类符：

- `dd` - 某月的天数。
- `mm` -某年的月数。

- `yy` - 2位数的年数。
- `yyyy` -4位数的年数。
- `ddd` - 某天的名字（缩略）。
- `mmm` - 某月份名字（缩略）。

所有其它的字符按1:1复制到输出字符串中。

Table 33-3表示一些可能的格式。 这将让你知道如何使用这些功能。所有输出行基于相同的日期：1959年11月23号。

Table 33-3. `PGTYPESdate_fmt_asc` 有效输入格式

格式	结果
`mmddyy`	`112359`
`ddmmyy`	`231159`
`yymmdd`	`591123`
`yy/mm/dd`	`59/11/23`
`yy mm dd`	`59 11 23`
`yy.mm.dd`	`59.11.23`
`.mm.yyyy.dd.`	`.11.1959.23.`
`mmm. dd, yyyy`	`Nov. 23, 1959`
`mmm dd yyyy`	`Nov 23 1959`
`yyyy dd mm`	`1959 23 11`
`ddd, mmm. dd, yyyy`	`Mon, Nov. 23, 1959`
`(ddd) mmm. dd, yyyy`	`(Mon) Nov. 23, 1959`

`PGTYPESdate_defmt_asc`

使用格式掩码转换C `char*` 到日期类型的值。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

函数接收一个指向保持操作(`d`)结果的日期值的指针， 解析日期(`fmt`)的格式掩码以及包含日期(`str`)文本表示的C char*字符串。 希望文本表示匹配格式掩码。但是你不需要字符串1:1映射到格式掩码。 这个函数仅分析相继顺序， 并且查找 `yy` 或者 `yyyy` 显示年的位置， `mm` 显示月的位置， `dd` 显示一天的位置。

Table 33-4表明一些可能的格式。 这将让你知道如果使用这个函数。

Table 33-4. `rdefmtdate` 有效输入格式

格式	字符串	结果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	在2525年，7月28号人类仍存活。	2525-07-28
dd-mm-yy	2525年7月28号	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

33.6.3. 时间戳类型

C中时间戳类型允许你的程序处理SQL类型时间戳数据。 参见Section 8.5 获取关于PostgreSQL服务器的等价类型。

下面的函数可以用于时间戳类型。

`PGTYPEStimestamp_from_asc`

将文本表示的时间戳解析成一个时间戳变量。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

函数接收一个解析(`str`)字符串和指向C char(*`endptr`*)指针。 此刻*ECPG*总是解析完整字符串， 因此它目前不支持存储`endptr `中第一无效字符地址。 你可以安全地设置` endptr`为空。

成功时函数返回解析的时间戳，产生错误时返回 `PGTYPESInvalidTimestamp` ， 并且设置 `errno` 为 `PGTYPES_TS_BAD_TIMESTAMP` 。 参见 *`PGTYPESInvalidTimestamp`* 获取这个值的重要注释。

一般情况下，输入的字符串可以包含一个所允许日期规范、 一个空格字符和允许的时间规范的任意组合。请注意，ECPG不支持时区。 它可以解析它们，但不适用于任何计算比如PostgreSQL服务器。 时区说明符默认是省略的。

Table 33-5包含输入字符串的一些例子。

Table 33-5. `PGTYPEStimestamp_from_asc` 有效输入格式

输入	结果
`1999-01-08 04:05:06`	`1999-01-08 04:05:06`
`January 8 04:05:06 1999 PST`	`1999-01-08 04:05:06`
`1999-Jan-08 04:05:06.789-8`	`1999-01-08 04:05:06.789 (忽略时区说明符)`
`J2451187 04:05-08:00`	`1999-01-08 04:05:00 (忽略时区说明符)`

`PGTYPEStimestamp_to_asc`

将日期转换成C char*字符串。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

函数接收时间戳 `tstamp` 作为其唯一的参数 并返回一个包含时间戳的文本表示的分配的字符串。

`PGTYPEStimestamp_current`

获取当前时间戳。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

该函数获取当前时间戳，并且将它保存到 `ts` 指向 的时间戳变量中。

`PGTYPEStimestamp_fmt_asc`

使用格式掩码将时间戳变量转换为C char*。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

这个函数接受一个指向时间戳转换为它的第一个参数(`ts`)的指针， 一个指向缓冲输出(`output`)， 最大长度已为输出缓冲区(`str_len`)分配， 并且为转换(`fmtstr`)设置掩码格式的指针。

一旦成功，该函数返回0，如果产生错误，则返回负值。

你可以为格式掩码使用以下的格式分类符。 格式分类符是和libc的 `strftime` 函数中使用的相同的。 任何非格式分类符将被复制到输出缓冲区。

- `%A` - 是由工作日全称的国家表示形式替换。
- `%a` - 是由工作日名称缩写的国家表示形式替换。
- `%B` - 是由月份名的全称的国家表示形式替换。
- `%b` - 是由月份名称缩写的国家表示形式替换。
- `%C` - 通过（年/100）作为十进制数替换；单位数前边加零。

- `%c` - 由时间和日期的国家表示形式替换。
- `%D` - 等同于 `%m/%d/%y` 。
- `%d` - 作为十进制数（01-31）按当月的一天替换。
- `%E*` `%O*` - POSIX区域扩展。 序列 `%Ec` `%EC` `%Ex` `%EX` `%Ey` `%EY` `%Od` `%Oe` `%OH` `%OI` `%Om` `%OM` `%OS` `%Ou` `%OU` `%OV` `%Ow` `%OW` `%Oy` 应该提供替代表示形式。

  此外，实现的 `%OB` 代表可选月份名字 （独立使用，没有提及天）。
- `%e` -作为十进制数（1-31）按当月的一天替换； 单位数前面有空格。
- `%F` - 等同于 `%Y-%m-%d` 。
- `%G` - 以每年作为一个世纪的十进制数替换。 今年是包含一周的大部分中的一个（星期一作为一周的第一天）。
- `%g` - 由 `%G` 中的同一年替换， 但作为一个没有世纪（00-99）的十进制数。
- `%H` - 作为十进制数（00-23）按小时（24小时）进行替换。
- `%h` -等同于 `%b` 。
- `%I` - 作为十进制数（01-12）按小时（12小时）进行替换。
- `%j` - 作为十进制数（001-366）按一年的一天来替换。
- `%k` - 作为十进制数（0-23）按小时（24小时）替换； 单位数前面有空格。
- `%l` - 作为十进制数（1-12）按小时（12小时）替换； 单位数前面有空格。
- `%M` - 作为十进制数（00-59）按分钟来替换。
- `%m` - 作为十进制数（01-12）按月替换。
- `%n` - 通过换行符替换。
- `%O*` - 等同于 `%E*` 。
- `%p` - 由合适的"午前"或"午后"的国家表示形式进行替换。
- `%R` - 等同于 `%H:%M` 。
- `%r` - 等同于 `%I:%M:%S%p`
- `%S` - 作为十进制数（00-60）按秒进行替换。
- `%s` -通过Epoch, UTC以来的秒数替换。
- `%T` - 等同于 `%H:%M:%S` 。

- `%t` -通过制表符替换。
- `%U` -按照十进制数（00-53）一年中的 周数取代（星期日作为一周的第一天）。
- `%u` - 按照十进制数（1-7）工 作日取代（星期一作为一周的第一天）。
- `%V` - 通过十进制数（01-53）一年 中的周数取代（星期一作为一周的第一天）。 如果在新的一年中包含一月一日的工作日有四天以上， 那么它是第1周；否则它是去年的最后一周，并且下一周是第1周。
- `%v` - 等同于 `%e-%b-%Y` 。
- `%W` - 通过十进制数（00-53） 一年的周数取代（星期一作为一周的第一天）。
- `%w` - 通过十进制数（0-6） 工作日取代（星期日作为一周的第一天）。
- `%X` - 通过时间的国家表示形式取代。
- `%x` - 通过日期的国家表示形式取代。
- `%Y` - 通过十进制数世纪年来取代。
- `%y` - 通过十进制数（00-99）没有世纪的年来取代。
- `%Z` - 由时区名称替换。
- `%z` -通过UTC时区偏移量取代； 前导加号为UTC东部，减号为UTC西部， 小时和分钟各跟随着两位数，它们之间没有分隔符 （常见的形式为RFC 822日期标题）。
- `%+` - 通过日期和时间的国家表示形式替换。
- `%-*` - GNU libc扩展。当执行数值输出时不要做任何填充。
- $_* - GNU libc扩展。明确声明空格填充。
- `%0*` - GNU libc扩展。明确声明零填充。
- `%%` - 通过 `%` 替换。

`PGTYPEStimestamp_sub`

从另外一个中减去一个时间戳，并且将结果保存在interval类型的变量中。

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数将减去时间戳变量，这个变量是 `ts2` 指向的 `ts1` 指向的时间戳的变量， 并将结果存储在 `iv` 指向的时间戳变量中。

成功时，函数返回0。如果发生错误则返回一个负值。

`PGTYPEStimestamp_defmt_asc`

从使用格式掩码的文本表示中分析一个时间戳值。

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

这个函数接受变量 `str` 时间戳的文本表示 以及格式掩码中使用的变量 `fmt` 。 结果将存储在 `d` 指向的变量中。

如果格式掩码 `fmt` 是空， 该函数将回落到缺省格式掩码 `%Y-%m-%d%H:%M:%S` 。

这是 *PGTYPEStimestamp_fmt_asc* 的反向函数。 参见文档找出可能的格式掩码项。

`PGTYPEStimestamp_add_interval`

增加interval变量到timestamp变量。

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

这个函数接受一个指向timestamp变量 `tin` 的指针， 一个指向interval变量 `span` 的指针。 它增加interval到timestamp， 并且将结果timestamp保存在 `tout` 指向的变量中。

成功时，这个函数返回0，如果产生错误，则返回一个负数。

`PGTYPEStimestamp_sub_interval`

从一个timestamp变量中减去interval变量。

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

这个函数减去interval变量，这个变量是 `span` 指向的 `tin` 指向的timestamp变量， 并且将结果存储在 `tout` 指向的变量中。

成功时，这个函数返回0。当产生错误的时候，返回负数。

33.6.4. 区间类型

C中区间类型允许你的程序处理SQL类型区间的数据。 参见Section 8.5获取 PostgreSQL服务器的等价类型。

下面的函数可以用于区间类型：

`PGTYPESinterval_new`

返回一个已分配的区间变量的指针。

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

释放已经分配区间变量的内存。

```
void PGTYPESinterval_new(interval *intvl);
```

`PGTYPESinterval_from_asc`

解析文本表示的区间。

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

该函数解析输入的字符串 `str` 并返回分配区间变量的指针。 此刻ECPG总是解析完整的字符串，所以目 前不支持存储在 `*endptr` 中的第一无效字符的地址。 你可以安全地设置 `endptr` 为空。

`PGTYPESinterval_to_asc`

将类型区间的变量转换成它的文本表示。

```
char *PGTYPESinterval_to_asc(interval *span);
```

该函数将转换 `span` 指向C char*的区间变量， 输出看起来像这样的例子： `@ 1 day 12 hours 59 mins 10 secs` 。

`PGTYPESinterval_copy`

复制区间类型的变量。

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

该函数复制 `intvlsrc` 指向 `intvldest` 指向的区间变量，注意， 你需要在目标变量前分配内存。

33.6.5. 十进制类型

decimal类型和numeric类型是类似的。然而， 它仅仅是一个30位数的最大精度。相反，numeric类型只能在堆上创建，decimal类型可以在栈或堆上创建 （通过函数 `PGTYPESdecimal_new` 和 `PGTYPESdecimal_free` ）。 Section 33.15中描述的 Informix兼容模式还有很多处 理decimal类型的其他函数。

下面的函数可以用于decimal类型，不仅包含在 `libcompat` 库中。

`PGTYPESdecimal_new`

请求一个新分配的decimal变量的指针。

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

任意decimal类型，释放所有内存。

```
void PGTYPESdecimal_free(decimal *var);
```

33.6.6. pgtypeslib的errno值

`PGTYPES_NUM_BAD_NUMERIC`

参数应该包含一个数值变量（或者指向一个数值变量） 但事实上内存中表示是无效的。

`PGTYPES_NUM_OVERFLOW`

发生溢出。因为numeric类型可以处理几乎任意精度， 将一个numeric变量转换为其它类型可能导致溢出。

`PGTYPES_NUM_UNDERFLOW`

发生下溢。因为numeric类型可以处理几乎任意精度， 将一个numeric变量转换为其它类型可能导致下溢。

`PGTYPES_NUM_DIVIDE_ZERO`

尝试除以零。

`PGTYPES_DATE_BAD_DATE`

无效的日期字符串被传递给 `PGTYPESdate_from_asc` 函数。

`PGTYPES_DATE_ERR_EARGS`

无效参数被传递给 `PGTYPESdate_defmt_asc` 函数。

`PGTYPES_DATE_ERR_ENOSHORTDATE`

通过 `PGTYPESdate_defmt_asc` 函数发现输入字符串中的无效标记。

`PGTYPES_INTVL_BAD_INTERVAL`

无效区间字符串被传递给 `PGTYPESinterval_from_asc` 函数， 或者无效区间值被传递给 `PGTYPESinterval_to_asc` 函数。

`PGTYPES_DATE_ERR_ENOTDMY`

在 `PGTYPESdate_defmt_asc` 函数中日/月/年分配不匹配。

`PGTYPES_DATE_BAD_DAY`

通过 `PGTYPESdate_defmt_asc` 函数发现某月值的无效天数。

`PGTYPES_DATE_BAD_MONTH`

通过 `PGTYPESdate_defmt_asc` 函数发现无效月数值。

`PGTYPES_TS_BAD_TIMESTAMP`

无效的timestamp字符串被传递给 `PGTYPEStimestamp_from_asc` 函数， 或者无效timestamp值被传递给 `PGTYPEStimestamp_to_asc` 函数。

`PGTYPES_TS_ERR_EINFTIME`

在环境中遇到的无限timestamp值不能处理它。

33.6.7. pgtypeslib的特殊常量

`PGTYPESInvalidTimestamp`

代表一个无效的时间戳的timestamp类型的值。 这是通过函数 `PGTYPEStimestamp_from_asc` 返回解析错误。 请注意，由于该 `timestamp` 数据类型的内部表示， 同时 `PGTYPESInvalidTimestamp` 也是一个有效的timestamp。 它设置 `1899-12-31 23:59:59` 。为了检测错误， 确保你的应用每次调用 `PGTYPEStimestamp_from_asc` 后 不仅测试 `PGTYPESInvalidTimestamp` 也能检测 `errno != 0` 。

33.7. 使用描述符范围

一个SQL描述符范围是处理 `SELECT`，`FETCH` 或者 `DESCRIBE` 语句结果的更复杂的方法。 一个SQL描述符范围把一行数据里的数据和元数据项组合到一个数据结构中。 元数据在执行动态SQL语句时特别有用，那里的结果列的属性可能不能提前知道。 PostgreSQL提供了两种使用描述符范围的方法： 命名的SQL描述符范围和C结构SQLDAs。

33.7.1. 命名SQL描述符范围

一个命名SQL描述符范围由一个头组成， 包含有关整个描述符的信息，一个或多个项描述符范围， 基本上每个描述结果行中的一个字段。

在你使用SQL描述符范围之前，你需要分配一个：

```
EXEC SQL ALLOCATE DESCRIPTOR _identifier_;
```

标示符用作描述符范围的"变量名"。 当你不再需要这个描述符，你应该释放它：

```
EXEC SQL DEALLOCATE DESCRIPTOR _identifier_;
```

要使用一个描述符范围，在一个 `INTO` 子句的存储目标里声明它， 而不是列出宿主变量：

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

如果结果集是空，描述符范围将包含来自查询的元数据，即字段名称。

为了尚未执行的预备查询， `DESCRIBE` 语句可用于获得结果集的元数据：

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

PostgreSQL 9.0之前， `SQL` 关键字是可选的， 所以使用 `DESCRIPTOR` 和 `SQL DESCRIPTOR` 产生命名SQL描述符范围。 现在，它是强制性的， 省略 `SQL` 关键词产生SQLDA描述符范围， 参阅Section 33.7.2。

在 `DESCRIBE` 和 `FETCH` 语句中， `INTO` 和 `USING` 关键字使用类似： 它们产生结果集合和描述符范围的元数据。

现在，我们应该如何从描述符范围里获取数据？ 你可以把描述符范围看作是一个有着命名字段的结构。 要从头检索字段数值并且把它存储到一个宿主变量里，使用下面的命令：

```
EXEC SQL GET DESCRIPTOR _name_ :_hostvar_ = _field_;
```

目前只定义了一个头字段： `_COUNT_`， 这个字段告诉我们有几个项描述符范围存在（也就是说，在结果里包含多少个字段）。 宿主变量需要是一个整数类型。要从项描述符范围里获取一个字段， 使用下面的命令：

```
EXEC SQL GET DESCRIPTOR _name_ VALUE _num_ :_hostvar_ = _field_;
```

`_num_` 可以是一个字符整数或者一个包含整数的宿主变量。 可能的字段有:

`CARDINALITY` (integer)

结果集中的行数

`DATA`

实际数据项（因此，这个字段的数据类型依赖于这个查询）

`DATETIME_INTERVAL_CODE` (integer)

当 `TYPE` 是 `9` 的时候，那么 `DATETIME_INTERVAL_CODE` 将有 `DATE` 的 `1` 值， `TIME` 的 `2` 值， `TIMESTAMP` 的 `3` 值， `TIME WITH TIME ZONE` 的 `4` 值或者 `TIMESTAMP WITH TIME ZONE` 的 `5` 值。

`DATETIME_INTERVAL_PRECISION` (integer)

未实现。

`INDICATOR` (integer)

描述符（标识一个空值或者一个截断的值）

`KEY_MEMBER` (integer)

未实现

`LENGTH` (integer)

字符中数据长度

`NAME` (string)

字段名称

`NULLABLE` (integer)

未实现

`OCTET_LENGTH` (integer)

字节数据的字符表示的长度

`PRECISION` (integer)

精度（类型 `numeric` ）

`RETURNED_LENGTH` (integer)

字符中数据长度

`RETURNED_OCTET_LENGTH` (integer)

字节数据的字符表示的长度

`SCALE` (integer)

比例（类型 `numeric` ）

`TYPE` (integer)

字段数据类型的数值代码

在 `EXECUTE` ，`DECLARE` 和 `OPEN` 语句中， `INTO` 和 `USING` 关键字的作用是不同的。 描述符范围可以手动的编译，为一个查询或者游标提供输入参数， 并且 `USING SQL DESCRIPTOR _name_` 是传递输入参数给一个参数化查询的一种方式。 编译命名SQL描述符范围的语句如下：

```
EXEC SQL SET DESCRIPTOR _name_ VALUE _num_ _field_ = :_hostvar_;
```

PostgreSQL支持检索更多的在一个 `FETCH` 语句中的记录和存储在宿主变量中的数据， 在这种情况下假设变量是一个数组。例如：

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

33.7.2. SQLDA描述符范围

SQLDA描述符范围是一个C语言结构， 它过去常常获取结果集和查询的元数据。 一个结构存储来自结果集中的一条记录。

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

注意省略 `SQL` 关键字。 Section 33.7.1中的 `INTO` 和 `USING` 关键字的 使用情况的段落有个例外， 也能适用于这里。在 `DESCRIBE` 语句中，如果使用了 `INTO` 关键字， 则 `DESCRIPTOR` 关键字完全省略。

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

使用SQLDA程序流是：

1. 准备一个查询，并且为它声明一个游标。
2. 为结果行声明SQLDA。
3. 为输入参数声明SQLDA，并且初始化它们(内存分配，参数设置)。
4. 打开具有输入SQLDA的游标
5. 从游标中抓取行，并且将它们存储到输出SQLDA中。
6. 从输出SQLDA中读取值到宿主变量中 (如果有必要使用转换)。
7. 关闭游标。
8. 自由内存区域分配给输入SQLDA。

33.7.2.1. SQLDA数据结构

SQLDA使用三个数据结构类型： `sqlda_t` ， `sqlvar_t` , 和 `struct sqlname` 。

> **Tip:** PostgreSQL的SQLDA与IBM DB2通用数据库中的一个有类似的数据结构。 所以DB2的SQLDA上的一些技术信息可以更好的帮助理解PostgreSQL的。

33.7.2.1.1. sqlda_t 结构

结构类型 `sqlda_t` 是实际SQLDA的类型。 它拥有一条记录。并且在链表中使用 `desc_next` 字段指针可以连接两个或更多个 `sqlda_t` 结构， 因此代表行的有序集合。因此，当抓取两个或更多行时， 应用程序通过每个 `sqlda_t` 节点 随后 `desc_next` 指针可以读取它们。

`sqlda_t` 的定义是：

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

该字段的意思是：

`sqldaid`

它包含文本字符串 `"SQLDA "` 。

`sqldabc`

它包含字节中分配空间的大小。

`sqln`

它包含一个参数化查询的情况下的输入参数数，使用 `USING` 关键字被传递给 `OPEN` ， `DECLARE` 或者 `EXECUTE` 语句。 在这种情况下它被作为 `SELECT` ， `EXECUTE` 或者 `FETCH` 语句的输出使用。 它的值和 `sqld` 语句是一样的。

`sqld`

它包含结果集中的字段数量。

`desc_next`

如果查询返回多条记录，那么返回多个链接SQLDA结构， 并且 `desc_next` 持有指向列表中下一项的指针。

`sqlvar`

这是结构集中列数组。

33.7.2.1.2. sqlvar_t结构

结构类型 `sqlvar_t` 持有列值和元数据比如类型和长度。 该类型的定义是：

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

该字段的含义是：

`sqltype`

包含该字段的类型标识符。 对于该值，参阅 `ecpgtype.h` 中的 `enum ECPGttype` 。

`sqllen`

包含该字段的二进制长度。比如4字节的 `ECPGt_int` 。

`sqldata`

指向该数据。关于数据的格式在Section 33.4.4 中描述。

`sqlind`

指向空指示器。0表示非空，-1表示空。

`sqlname`

该字段名称。

33.7.2.1.3. struct sqlname结构

`struct sqlname` 结构持有列名。 它作为 `sqlvar_t` 结构成员被使用。 该结构定义是：

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

该字段含义是：

`length`

包含该字段名长度。

`data`

包含实际字段名。

33.7.2.2. 使用SQLDA检索结果集

通过SQLDA检索查询结果集的一般步骤是：

1. 声明 `sqlda_t` 结构用来接收结果集。
2. 执行 `FETCH` / `EXECUTE` / `DESCRIBE` 命令用来处理指定已声明SQLDA的查询。
3. 通过查看 `sqln` 检查结果集中的记录数， `sqlda_t` 结构成员。

4. 从 `sqlvar[0]`， `sqlvar[1]` 等中获得每列的值， `sqlda_t` 结构成员
5. 通过 `desc_next` 指针转到下一行(`sqlda_t` 结构)， `sqlda_t` 结构成员。
6. 你需要重复以上步骤

这是一个通过SQLDA检索结果集的例子。

首先，声明一个 `sqlda_t` 结构以接收结果集。

```
sqlda_t *sqlda1;
```

接下来，在命令中声明SQLDA。这是 `FETCH` 命令实例。

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

在链接表后运行循环以检索行。

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

在循环中，运行另外一个循环以检索行中的每列数据（ `sqlvar_t` 结构）。

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

为了得到列值，检查 `sqltype` 值， `sqlvar_t` 结构成员。然后， 切换适当方式，依赖于列类型，从宿主变量 `sqlvar` 字段拷贝数据。

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqll
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

33.7.2.3. 使用SQLDA传递查询参数

使用SQLDA传递输入参数给预备查询的一般步骤是：

1. 创建预备查询(预备语句)
2. 作为输入SQLDA声明sqlda_t结构。
3. 为了输入SQLDA分配内存区域(作为sqlda_t结构)。
4. 在已分配内存中设置（拷贝）输入值。
5. 打开具有声明输入SQLDA的游标。

这有个例子。

首先，创建一个预备语句。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid =
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

下一步，为SQLDA分配内存，并且在 `sqln` 中设置输入参数数， `sqlda_t` 结构成员变量。 当预备查询需要两个或更多个输入参数的时候，应用程序必须分配额外内存空间，它 是通过(nr. of params - 1) * sizeof(sqlvar_t)计算的。 这里显示的是为两个输入参数分配内存空间的例子。

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* number of input variables */
```

内存分配后，存储参数值到 `sqlvar[]` 数组。 （当该SQLDA正在接收结果集时，这是用于检索列值的相同数组。） 在这个例子中，输入参数是有字符串类型的 `"postgres"` ， 以及有整数类型的 `1` 。

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

打开游标并且声明事先准备的SQLDA，将输入参数传递给预备语句。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

最后，使用输入SQLDA之后，必须显式释放已分配内存空间，不像用于接收查询 结果的SQLDA。

```
free(sqlda2);
```

33.7.2.4. 使用SQLDA示例应用程序

这是一个示例程序，描述了如何获取数据库访问统计，通过输入参数声明，来自系统表。

这个应用程序连接两个系统表，数据库OID上的pg_database和 pg_stat_database，并且读取、显示由两个输入参数（ `postgres` 和OID `1` ）检索的数据库统计。

首先，为输入声明SQLDA，以及为输出声明SQLDA。

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /*输出描述符*/
sqlda_t *sqlda2; /*输入描述符*/
```

下一步，连接数据库，准备语句，并且为预备语句声明游标。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

接下来，为输入参数将一些值放在输入SQLDA中。为输入SQLDA分配内存，并且设置 输入参数数到 `sqln` 。存储类型，值以及值长度到 `sqltype` ， `sqldata` 中， 并且将 `sqllen` 放在 `sqlvar` 结构中。

```
/*为输入参数创建SQLDA结构 */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /*输入变量数*/

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

在建立输入SQLDA后，打开具有输入SQLDA的一个游标。

```
/*打开具有输入参数的游标。*/

    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

从已打开的游标中读取行到输出SQLDA。（一般来说，你必须在循环中反复调用 `FETCH` ， 为了读取结果集中的所有行。）

```
    while (1)
    {
        sqlda_t *cur_sqlda;

/*分配描述符给游标*/

        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

接下来，从SQLDA中检索已读取记录，通过下面 `sqlda_t` 结构中 的连接表。

```
    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

读取第一条记录中的每一列。列数被存储在 `sqld` 中， 第一列的实际数据被存储在 `sqlvar[0]` ， `sqlda_t` 结构的两个 成员中。

```
/* 输出行中每一列*/
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

目前，该列数据被存储在变量 `v` 中。拷贝每个数据到宿主变量， 为了列类型查看 `v.sqltype` 。

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf
                    break;

                case ECPGt_int: /* integer */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

在处理完所有记录之后关闭游标，并且断开数据库连接。

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

在Example 33-1中显示了整个程序。

Example 33-1. SQLDA程序示例

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /*输出描述符*/
sqlda_t *sqlda2; /*输入描述符*/
```

```
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

/*为输入参数创建SQLDA结构*/

    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* a number of input variables */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

/*打开具有输入参数的游标*/

    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

/*分配描述符给游标*/
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

/*输出行中每一列*/
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ? sizeof(var_b
                        break;

                    case ECPGt_int: /* integer */
                        memcpy(&intval, sqldata, sqllen);
```

```
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char) sql
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

该例子输出应该看起来像下面这样（一些数字有所不同）。

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

33.8. 错误处理

本节描述了如何处理异常情况以及嵌入SQL程序的警告。有两个非排他性功能可以解决。

- 配置回调用来处理警告以及使用 `WHENEVER` 命令处理错误条件。
- 关于错误或者警告的详细信息可以从 `sqlca` 变量中获得。

33.8.1. 设置回调

当产生特定条件时，捕获错误和警告的一个简单方法是设置一个要执行的具体操作。通常：

```
EXEC SQL WHENEVER _condition_ _action_;
```

`_condition_` 可以是下列之一：

`SQLERROR`

当在SQL语句执行期间发生错误时，调用指定操作。

`SQLWARNING`

当在SQL语句执行期间发生警告时，调用指定操作。

`NOT FOUND`

当SQL语句检索或者影响零行，则调用指定操作。（这个条件不是错误， 但是你可能对特意处理它感兴趣。）

`_action_` 可以是下列之一：

`CONTINUE`

这实际上意味着该条件被忽略。这是缺省的。

`GOTO` `_label_``GO TO` `_label_`

跳转到指定标签（使用C `goto` 语句）。

`SQLPRINT`

输出标准错误信息。这对于简单程度或者原型期间非常有用。 不能配置该信息的详细信息。

`STOP`

调用 `exit(1)` ，这将终止程序。

`DO BREAK`

执行C语句 `break` 。这只有在循环中或者 `switch` 语句中使用。

`CALL` `_name_` (`_args_`) `DO` `_name_` (`_args_`)

调用具有指定参数的指定C函数。

SQL标准仅仅提供 `CONTINUE` 和 `GOTO` (和 `GO TO` ）操作。

下面是一个你可能想在简单程序中使用的例子。当发生警告以及发生错误终止程序时， 它输出一个简单消息：

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

语句 `EXEC SQL WHENEVER` 是SQL预处理器的指令。 而不是C语句。 错误或者警告操作设置处理器出现的地方中适用的所有嵌入SQL语句。 除非在第一个 `EXEC SQL WHENEVER` 和产生条件的SQL语句之间 为同一条件设置不同的操作，不管C程序中的控制流。 所以下面两个C程序片段都不会产生期望效果：

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

33.8.2. sqlca

为了更强大的错误处理，嵌入SQL接口提供了使用下列结构的名字 `sqlca` （SQL通信区） 的全局变量。

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（在一个多线程程序中，每一个线程自动获取 `sqlca` 的拷贝。该工作类似于标准C全局变量 `errno` 的处理。）

`sqlca` 涵盖警告和错误。如果在语句执行期间发生 多个警告和错误，那么 `sqlca` 将只包含最后一个信息。

如果在最后SQL语句没有发生错误，则 `sqlca.sqlcode` 为0， `sqlca.sqlstate` 是 `"00000"` 。如果发生了警告或者错误，那么 `sqlca.sqlcode` 是负数并且 `sqlca.sqlstate` 不同于 `"00000"` 。正数 `sqlca.sqlcode` 表示无害条件，比如最后查询返回零行。`sqlcode` 和 `sqlstate` 是两个 不同的错误编码方案；详情如下。

如果最后一个SQL语句成功了，那么 `sqlca.sqlerrd[1]` 包含处理行的OID，如果适用，则 `sqlca.sqlerrd[2]` 包含处理或返回行的行数，如果适用该命令。

在错误或警告的情况下， `sqlca.sqlerrm.sqlerrmc` 将包含描述错误的字符串。字段 `sqlca.sqlerrm.sqlerrml` 包含存储在 `sqlca.sqlerrm.sqlerrmc` （ `strlen()` 的结果，C程序员不感兴趣）中的错误信息。 注意一些消息太长而不适合固定大小的 `sqlerrmc` 数组； 它们将被截断。

在一个警告的情况下， `sqlca.sqlwarn[2]` 设置为 `W` 。（在所有其他情况下，它被设置为不同于 `W` 的东西。）如果 `sqlca.sqlwarn[1]` 被设置为 `W` ，那么一个值被存储在宿主变量的时候，截断它。 如果任何其他元素设置为显示一个警告，则 `sqlca.sqlwarn[0]` 设置为 `W` 。

字段 `sqlcaid` ， `sqlcabc` ， `sqlerrp` ， 以及 `sqlerrd` 和 `sqlwarn` 的剩余元素 目前没有任何有用信息。

在SQL标准中没有定义结构 `sqlca` ， 但是在其他几个SQL数据库系统中实现了。定义核心是相似的，但是如果你想要 编写可移植应用程序，那么你应该仔细调查不同的实现。

这是一个结合 `WHENEVER` 和 `sqlca` 的使用的例子， 当发生错误时，输出 `sqlca` 的内容。 在安装更多"user-friendly"错误处理程序之前， 这可能用于调试或者原型应用。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n", sqlca.sqlerrd[0],sqlca.sqlerrd[
                                                          sqlca.sqlerrd[3],sqlca.sqlerrd[
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0], sqlca.sqlwarn
                                                          sqlca.sqlwarn[3], sqlca.sqlwarn
                                                          sqlca.sqlwarn[6], sqlca.sqlwarn
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

结果可能如下所示（这里错误归因于表名字拼写错误）：

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

33.8.3. `SQLSTATE` vs. `SQLCODE`

字段 `sqlca.sqlstate` 和 `sqlca.sqlcode` 是提供错误码的两个不同模式。 两者来自SQL标准，但是 `SQLCODE` 在标准SQL-92 版本中已经过时，并且在后期版本中已经废除。因此， 强烈建议新应用使用 `SQLSTATE` 。

`SQLSTATE` 是五字符数组。 五字符包含数字或者表示不同错误和警告条件代码的大写字母。 `SQLSTATE` 有一个分层模式： 前两个字符表示条件的一般类， 最后三个字符表示一般条件的子类。 通过代码 `00000` 表示成功状态。 `SQLSTATE` 代码是SQL标准中定义最多部分。 PostgreSQL服务器本地支持 `SQLSTATE` 错误代码；因此通过在所有应用程序中 使用该错误代码方案实现高度一致性。 更多信息参阅Appendix A。

`SQLCODE` ，已废弃的错误编码方案，是一个简单的integer。 0值表示成功，正值表示额外信息成功， 负值表示错误。 SQL标准仅仅定义正值+100， 这表示返回最后命令或者影响零行，并且 没有明确负值。因此，该方案实现差的移植性，而且没有分层编码安排。 从历史角度，PostgreSQL嵌入的SQL预处理器 为它的使用分配了一些指定 `SQLCODE` 。 使用数值和符号名称将它列在下面。记住这些是不能移植到其他SQL实现的。 为了简化应用程序移植到 `SQLSTATE` 方案，相应的 `SQLSTATE` 也被列出来。然而， 在两个方案（实际上是多对多）之间没有一对一或者一对多映射， 因此在每种情况下你应该咨询列在Appendix A中的全球 `SQLSTATE` 。

这些是已分配的 `SQLCODE` 值：

0 (`ECPG_NO_ERROR`)

表明没有错误。(SQLSTATE 00000)

100 (`ECPG_NOT_FOUND`)

这是无害条件表明检索最后一条命令或者处理零行，或者你在游标结尾。(SQLSTATE 02000)

当在循环中处理游标时，你可以使用该代码作为检测什么时候终止循环的方式，像这样：

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

但是 `WHENEVER NOT FOUND DO BREAK` 有效的内部执行这个，因此 在明确写这个时通常没有优势。

-12 (`ECPG_OUT_OF_MEMORY`)

表明耗尽了你的虚拟内存。作为 `-ENOMEM` 定义该数值。 (SQLSTATE YE001)

-200 (`ECPG_UNSUPPORTED`)

表明预处理器产生了该库不知道的一些东西。可能你正在该预处理器和该库不兼容版本上运行。(SQLSTATE YE002)

-201 (`ECPG_TOO_MANY_ARGUMENTS`)

这意味着指定命令比期望命令宿主变量更多。(SQLSTATE 07001或者07002)

-202 (`ECPG_TOO_FEW_ARGUMENTS`)

这意味着指定命令比期望命令宿主变量更少。(SQLSTATE 07001或者07002)

-203 (`ECPG_TOO_MANY_MATCHES`)

这意味着查询返还多行但是语句只准备存储一个结果行（比如， 因为指定变量不是数组）。(SQLSTATE 21000)

-204 (`ECPG_INT_FORMAT`)

宿主变量是类型 `int` ，并且数据库中数据是不同类型，而且 包含不能解释为 `int` 类型的值。为这种转换该库使用 `strtol()` 。(SQLSTATE 42804)

-205 (`ECPG_UINT_FORMAT`)

宿主变量是类型 `无符号int` ，并且数据库中数据是不同类型，而且 包含不能解释为 `无符号int` 类型的值。 为这种转换该库使用 `strtoul()` 。(SQLSTATE 42804)

-206 (`ECPG_FLOAT_FORMAT`)

宿主变量是类型 `float` ，并且数据库中数据是另一种类型，而且 包含不能解释为 `float` 类型的值。 为这种转换该库使用 `strtod()` 。(SQLSTATE 42804)

-207 (`ECPG_NUMERIC_FORMAT`)

宿主变量是类型 `numeric` ，并且数据库中数据是另一种类型，而且 包含不能解释为 `numeric` 类型的值。(SQLSTATE 42804)

-208 (`ECPG_INTERVAL_FORMAT`)

宿主变量是类型 `interval` ，并且数据库中数据是另一种类型，而且 包含不能解释为 `interval` 类型的值。(SQLSTATE 42804)

-209 (`ECPG_DATE_FORMAT`)

宿主变量是类型 `date` ，并且数据库中数据是另一种类型，而且 包含不能解释为 `date` 类型的值。(SQLSTATE 42804)

-210 (`ECPG_TIMESTAMP_FORMAT`)

宿主变量是类型 `timestamp` ，并且数据库中数据是另一种类型，而且 包含不能解释为 `timestamp` 类型的值。(SQLSTATE 42804)

-211 (`ECPG_CONVERT_BOOL`)

这意味着宿主变量是类型 `bool` ， 并且数据库中数据既不是 `'t'` 也不是 `'f'` 。(SQLSTATE 42804)

-212 (`ECPG_EMPTY`)

发送到PostgreSQL服务器的语句是空的。 （这通常不会发生在嵌入SQL程序中，因此它可能指向一个内部错误。） (SQLSTATE YE002)

-213 (`ECPG_MISSING_INDICATOR`)

返回一个空值，而且没有提供空指示符变量。(SQLSTATE 22002)

-214 (`ECPG_NO_ARRAY`)

一个普通变量被用于需要数组的地方。(SQLSTATE 42804)

-215 (`ECPG_DATA_NOT_ARRAY`)

数据库返回需要数组值位置的普通变量。(SQLSTATE 42804)

-220 (`ECPG_NO_CONN`)

该程序试图访问一个不存在的连接。(SQLSTATE 08003)

-221 (`ECPG_NOT_CONN`)

该程序试图访问一个存在但无法打开的连接。（这是一个内部错误。）(SQLSTATE YE002)

-230 (`ECPG_INVALID_STMT`)

你正尝试使用的语句未准备好。(SQLSTATE 26000)

-239 (`ECPG_INFORMIX_DUPLICATE_KEY`)

重复键错误，违反唯一约束（Informix兼容模式）。(SQLSTATE 23505)

-240 (`ECPG_UNKNOWN_DESCRIPTOR`)

未找到指定描述符。你尝试使用的语句未准备好。(SQLSTATE 33000)

-241 (`ECPG_INVALID_DESCRIPTOR_INDEX`)

指定的描述符索引超出了范围。(SQLSTATE 07009)

-242 (`ECPG_UNKNOWN_DESCRIPTOR_ITEM`)

请求无效描述符项。（这是个内部错误。） (SQLSTATE YE002)

-243 (`ECPG_VAR_NOT_NUMERIC`)

在动态语句执行的过程中，数据库返回一个数值，但宿主变量不是数字的。 (SQLSTATE 07006)

-244 (`ECPG_VAR_NOT_CHAR`)

在动态语句执行的过程中，数据库返回一个非数值， 但宿主变量是数字的。 (SQLSTATE 07006)

-284 (`ECPG_INFORMIX_SUBSELECT_NOT_ONE`)

子查询结果不是单行（Informix兼容模式）。(SQLSTATE 21000)

-400 (`ECPG_PGSQL`)

PostgreSQL服务器产生一些错误。 包含的错误消息来自PostgreSQL服务器。

-401 (`ECPG_TRANS`)

PostgreSQL发出信号我们不能启动，提交，或者回滚事务。 (SQLSTATE 08007)

-402 (`ECPG_CONNECT`)

尝试与数据库的连接没有成功。(SQLSTATE 08001)

-403 (`ECPG_DUPLICATE_KEY`)

重复键错误，违反唯一约束。(SQLSTATE 23505)

-404 (`ECPG_SUBSELECT_NOT_ONE`)

子查询结果不是单行。(SQLSTATE 21000)

-602 (`ECPG_WARNING_UNKNOWN_PORTAL`)

指定一个无效游标名。(SQLSTATE 34000)

-603 (`ECPG_WARNING_IN_TRANSACTION`)

事务正在进行中。(SQLSTATE 25001)

-604 (`ECPG_WARNING_NO_TRANSACTION`)

这是一个非活跃（进行中）事务。(SQLSTATE 25P01)

-605 (`ECPG_WARNING_PORTAL_EXISTS`)

指定一个已经存在游标名。(SQLSTATE 42P03)

33.9. 预处理器指令

可用的几种预处理器指令， 它修改 `ecpg` 预处理器分析方式以及处理文件方式。

33.9.1. 包含文件

为了包含一个外部文件到你的嵌入SQL程序中，使用：

```
EXEC SQL INCLUDE _filename_;
EXEC SQL INCLUDE <_filename_>;
EXEC SQL INCLUDE "_filename_";
```

嵌入的SQL预处理器将寻找名为 `_filename_` .h的文件， 处理它，并且将它包含在产生的C输出中。因此，正确处理包含文件中的嵌入SQL语句。

`ecpg` 预处理器将按照下面顺序在几个目录中 搜索文件：

- 当前目录
- `/usr/local/include`
- PostgreSQL包含目录，在编译时定义 (比如 `/usr/local/pgsql/include`)
- `/usr/include`

但是当使用 `EXEC SQL INCLUDE "``_filename_` "时，仅仅搜索当前目录。

在每个目录中，预处理器将首先寻找给定的文件名，如果没有找到将追加 `.h` 到文件名然后再次尝试（除非指定文件名已经有这种后缀）。

注意 `EXEC SQL INCLUDE` 是不一样的：

```
#include <_filename_.h>
```

因为这个文件不受SQL命令预处理的影响。当然，你可以继续使用 包含其他头文件的C `#include` 指令。

> **Note:** 包含文件名大小写敏感，即使其余的 `EXEC SQL INCLUDE` 命令遵循正常的SQL大小写敏感规则。

33.9.2. define和undef指令

类似于C中 `#define` 指令，嵌入的SQL有一个类似概念：

```
EXEC SQL DEFINE _name_;
EXEC SQL DEFINE _name_ _value_;
```

所以你可以定义一个名字：

```
EXEC SQL DEFINE HAVE_FEATURE;
```

你也可以定义常数：

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

使用 `undef` 删除以前的定义：

```
EXEC SQL UNDEF MYNUMBER;
```

当然你可以继续在你的嵌入SQL程序中使用C版本 `#define` 和 `#undef` 。不同的是你定义值的评估不同。如果 你使用 `EXEC SQL DEFINE` ， 那么 `ecpg` 预处理器评估定义且替换该值。比如如果你写：

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

那么 `ecpg` 将执行替换， 而且你的C编译器不会看到任何名字或者标示符 `MYNUMBER` 。 注意你不能为打算用在嵌入SQL查询中的常数使用 `#define` ， 因为在这种情况下嵌入的SQL预编译器不能看到这个声明。

33.9.3. ifdef，ifndef，else，elif和endif指令

你可以使用下面指令有条件地编译代码段：

`EXEC SQL ifdef` `_name_`;

如果 `_name_` 已经和 `EXEC SQL define` `_name_` 被创建，那么检查 `_name_` 并处理 随后行。

`EXEC SQL ifndef` `_name_`;

如果 `_name_` 和 `EXEC SQL define` `_name_` 没有被创建， 那么检查 `_name_` 并处理 随后行。

`EXEC SQL else;`

开始处理另一部分到 `EXEC SQL ifdef` `_name_` 或者 `EXEC SQL ifndef` `_name_` 介绍的部分。

`EXEC SQL elif` `_name_`;

如果 `_name_` 和 `EXEC SQL define` `_name_` 已经被创建，那么检查 `_name_` 并且开始另一部分。

`EXEC SQL endif;`

结束另一部分。

例子：

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

33.10. 处理嵌入的SQL程序

现在你已知道如何形成嵌入SQL C程序，你可能想要知道如何编译它们。 编译之前你通过嵌入的SQL C预处理器运行文件，这将转换你用于特定函数调用的 SQL 语句。编译完之后，你必须连接包含所需函数的特殊库。这些函数 从参数中抓取信息，使用libpq接口执行SQL 命令，将结果放在输出指定的参数中。

预处理程序被称为 `ecpg`， 包含在正常PostgreSQL安装中。 嵌入的SQL程序通常以扩展名 `.pgc` 命名。 如果你有 `prog1.pgc` 程序文件，你可以通过简单调用处理它。

```
ecpg prog1.pgc
```

这将创建名为 `prog1.c` 的文件。如果你的输入文件 不遵循建议的命名模式，你可以使用 `-o` 选项明确的指定输出文件。

预处理文件可以正常编译，比如：

```
cc -c prog1.c
```

生成的C源文件包含来自PostgreSQL安装的头文件，因此 如果你在缺省不被搜索的地方安装PostgreSQL，那么 你必须添加选项比如 `-I/usr/local/pgsql/include` 到 编译命令行。

为了连接嵌入式SQL程序，你需要包含 `libecpg` 库，像这样：

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

再次，你可能需要添加像 `-L/usr/local/pgsql/lib` 到该命令行的选项。

你可以使用 `pg_config` 或者 `pkg-config` 以及包名 `libecpg` 以 获得安装目录。

如果你使用make管理大对象编译过程， 它可能方便地包含下面你的makefiles中的隐式规则：

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

`ecpg` 命令完整语法在ecpg中有详细描述。

ecpg库缺省是线程安全的。然而，你可能需要 使用一些线程命令行选项来编译你的客户端代码。

33.11. 库函数

`libecpg` 库主要包含"隐藏的"函数， 它用于实现嵌入SQL命令表达的功能。但是有一些可以直接调用的函数。 注意这使得您的代码可移植。

- 如果调用第一个非零参数， `ECPGdebug(int` *on* `, FILE *` *stream* `)` 打开调试日志。 调试日志在 `_stream_` 上执行。 日志包含带有插入的所有输入变量的SQL语句，以及 来自PostgreSQL服务器的结果。 当搜索SQL语句中的错误时，这是非常有用的。

  > **Note:** 在Windows上，如果ecpg库和应用程序是以不同标识被编译，那么 该函数调用将崩溃，因为 `FILE` 指针内部表示形式不同。 特别是，多线程/单线程，释放/调试，以及静态/动态信号 对于该库以及使用该库的所有应用程序是一样的。

- `ECPGget_PGconn(const char *` *connection_name* `)` 返回由给定名称标识的数据库连接句柄。 如果 `_connection_name_` 设置为 `NULL` ， 则返回当前连接句柄。如果没有连接句柄可以被识别， 则该函数返回 `NULL` 。如果必要的话，返回的连接句柄可以用于 从libpq调用任何其他函数。

  > **Note:** 操作随着libpq例程由ecpg直接组成的数据库连接句柄是个坏主意。

- `ECPGtransactionStatus(const char *` *connection_name* `)` 返回通过 `_connection_name_` 标识的给定连接的当前事务状态。 参阅Section 31.2和libpq的 `PQtransactionStatus()` 获取关于返回状态码的详细信息。

- 如果你连接到一个数据库，则 `ECPGstatus(int` *lineno* `, const char*` *connection_name* `)` 返回真，否则返回假。 如果正在使用一个连接，则 `_connection_name_` 是 `NULL` 。

33.12. 大对象

ECPG不直接支持大对象，但是ECPG应用可以通过libpq大对象函数操作大对象， 通过调用 `ECPGget_PGconn()` 函数获取必要的 `PGconn` 对象。 （然而，应该小心使用 `ECPGget_PGconn()` 函数 以及直接接触 `PGconn` 对象，理想情况下不与其他ECPG数据库访问调用混合）。

参阅Section 33.11获取关于 `ECPGget_PGconn()` 的更多信息。 关于大对象函数接口的更多信息，参阅Chapter 32。

在事务块中必须调用大对象函数，因此当关闭自动提交时，那么必须明确发出 `BEGIN` 命令。

Example 33-2显示了在ECPG应用中如何创建，写，读取大对象的示例程序。

Example 33-2. ECPG程序访问大对象

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* create */
    loid = lo_create(conn, 0);
    if (loid < 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* write test */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc < 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
```

```
    if (rc < 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc < 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc < 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* check */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* cleanup */
    rc = lo_unlink(conn, loid);
    if (rc < 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.13. C++应用程序

ECPG对C++应用程序有一些有限的支持。本节介绍一些注意事项。

`ecpg` 预处理程序采取写入C（或者类似C）的输入文件，并且 嵌入SQL命令，将嵌入SQL命令转换为C语言块， 最后生成 `.c` 文件。 当在C++中使用时， 通过 `ecpg` 产生的C语言块使用的库函数的头文件声明 被包裹在 `extern "C" { ... }` 块中。 因此他们应该在C++中无缝工作。

一般情况下，然而， `ecpg` 预处理器仅仅了解C；它 不处理特殊语法并且保留C++语言关键字。因此， 写入使用复杂特定C++功能的C++应用程序代码的一些嵌入SQL代码可能 不能正确地被预处理或者可能不会按预期的工作。

在C++应用程序中使用嵌入SQL代码的安全方式是在C模块中隐藏ECPG调用， 其中C++应用程序代码调用访问数据库，并且连同C++代码其余部分一起连接。 参阅Section 33.13.2获取关于它的更多信息。

33.13.1. 宿主变量范围

`ecpg` 预处理器理解C中变量范围。在C语言中， 这是简单地因为变量范围基于他们的代码块。在C++中， 然而， 类成员变量参考来自声明位置的不同代码块。 因此 `ecpg` 预处理程序不理解类成员变量的范围。

比如，在下面情况下， `ecpg` 预处理器无法找到 `test` 方法中变量 `dbname` 的任何声明， 因此产生错误。

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

这个代码将产生类似这样的错误。

```
<kbd class="literal">ecpg test_cpp.pgc</kbd>
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

为了避免这个范围问题， `test` 方法可以改为使用局部变量作为 中间存储器。但是这个方法仅仅是一个低劣的解决办法，因为 它丑化代码并且降低性能。

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

33.13.2. C++应用程序开发与外部C模块

如果你理解C++中 `ecpg` 预处理器的这些技术局限性， 你可能得到这样的结论在链接阶段链接中C对象与C++对象使得C++应用程序 使用ECPG功能可能好于在C++代码中直接写一些嵌入SQL命令。

已经创建三种文件：一个C文件(`*.pgc`)， 头文件和C++文件：

`test_mod.pgc`

执行SQL命令的子程序模块嵌入C中。它将通过预处理器被转换为 `test_mod.c` 。

```
#include "test_mod.h"
#include &lt;stdio.h&gt;

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

`test_mod.h`

C模块中(`test_mod.pgc`)使用函数声明的头文件通过 `test_cpp.cpp` 被包含。 这个文件在声明周围有一个 `extern "C"` 块，因为 它将从C++模块链接。

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

`test_cpp.cpp`

该应用程序主要代码，包含 `main` 程序和例子中C++类。

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t-&gt;test();
    return 0;
}
```

为了编译应用程序，如下进行。 通过运行 `ecpg` ， 转换 `test_mod.pgc` 到 `test_mod.c` ， 使用C编译器通过编译 `test_mod.c` 产生 `test_mod.o` 。

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

下一步，使用C++编译器通过编译 `test_cpp.cpp` 、 生成 `test_cpp.o` 。

```
c++ -c test_cpp.cpp -o test_cpp.o
```

最后，链接这些对象文件， `test_cpp.o` 和 `test_mod.o` 到一个可执行文件中，使用C++编译器驱动：

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

33.14. 嵌入的SQL命令

Table of Contents

本节描述的所有SQL命令仅限于嵌入的SQL。参阅 列在Reference I, *SQL 命令*中的SQL命令， 这也可以用于嵌入的SQL，除非另有说明。

ALLOCATE DESCRIPTOR

Name

ALLOCATE DESCRIPTOR -- 分配一个SQL描述区

Synopsis

```
ALLOCATE DESCRIPTOR _name_
```

描述

`ALLOCATE DESCRIPTOR` 分配一个新命名SQL描述符区域， 可以用于改变PostgreSQL服务器和主机程序之间的数据。

使用 `DEALLOCATE DESCRIPTOR` 命令之后应该释放描述符区域。

参数

`_name_`

SQL描述符名字区分大小写。这可以是SQL标识符或者宿主变量。

例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

兼容性

在SQL标准中声明 `ALLOCATE DESCRIPTOR` 。

参见

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

Name

CONNECT -- 建立数据库连接

Synopsis

```
CONNECT TO _connection_target_ [ AS _connection_name_ ] [ USER _connection_user_name_ ]
CONNECT TO DEFAULT
CONNECT _connection_user_name_
DATABASE _connection_target_
```

描述

`CONNECT` 命令在客户端和PostgreSQL服务器之间 创建了连接。

参数

`_connection_target_`

`_connection_target_` 在几种形式之一上指定了连接的目标服务器。

[`_database_name_`] [`@``_host_` ] [ `:``_port_`]

TCP/IP连接

`unix:postgresql://``_host_` [ `:``_port_` ] `/` [ `_database_name_` ] [ `?``_connection_option_`]

连接Unix-域套接字

`tcp:postgresql://``_host_` [ `:``_port_` ] `/` [ `_database_name_` ] [ `?``_connection_option_`]

TCP/IP连接

SQL字符串常量

在上述形式之一中包含一个值

宿主变量

类型 `char[]` 或者 `VARCHAR[]` 的宿主变量包含上述形式之一的值。

`_connection_object_`

用于连接可选标识符，这样就可以指向其他命令。 这可以是一个SQL标识符或者宿主变量。

`_connection_user_`

用于数据库连接的用户名。

该参数可以指定用户名和密码，使用下面形式之一 `_user_name_` / `_password_` , `_user_name_` IDENTIFIED BY `_password_` 或者 `_user_name_` USING `_password_` .

用户名和密码可以是SQL标识符，字符串常量，或者宿主变量。

`DEFAULT`

使用所有缺省连接参数，正如通过libpq定义的。

例子

指定连接参数的几种变形：

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY c
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED B
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connect
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connect
```

这里有一个例子程序阐述了使用宿主变量指定连接参数：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;

char *dbname     = "testdb";    /*数据库名称*/
    char *user       = "testuser";  /*连接用户名*/
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* 连接字符串 */
    char ver[256];                  /*存储版本字符串的缓冲区*/
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

兼容性

在SQL标准中声明 `CONNECT`， 但是 连接参数的格式是具体实施的。

参见

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

Name

DEALLOCATE DESCRIPTOR -- 重新分配SQL描述符区域

Synopsis

```
DEALLOCATE DESCRIPTOR _name_
```

描述

`DEALLOCATE DESCRIPTOR` 重新分配命名的SQL描述符区域。

参数

`_name_`

描述符名字将重新被分配。它对大小写敏感。这可以是一个SQL标识符 或者宿主变量。

例子

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

在SQL标准中声明 `DEALLOCATE DESCRIPTOR` 。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

Name

DECLARE -- 定义游标

Synopsis

```
DECLARE _cursor_name_ [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITH
DECLARE _cursor_name_ [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITH
```

描述

`DECLARE` 为了迭代预备语句结果集声明了游标。 该命令与直接SQL命令 `DECLARE` 略微有些不同语法： 后者执行查询并且为检索准备结果集，这个嵌入SQL命令只为迭代查询结果集 声明作为"循环变量"的名字； 当使用 `OPEN` 命令打开游标时，发生实际执行情况。

参数

`_cursor_name_`

游标名大小写敏感。这可以是一个SQL标识符或者宿主变量。

`_prepared_name_`

一个准备好查询的名字，要么作为SQL标识符或者宿主变量。

`_query_`

SELECT或者VALUES命令 将提供通过游标返回的行。

关于游标选项的含义，参阅DECLARE。

例子

为查询声明游标的例子：

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

为预备语句声明游标的例子：

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

兼容性

在SQL标准中声明 `DECLARE` 。

参见

OPEN, CLOSE, DECLARE

DESCRIBE

Name

DESCRIBE -- 获得关于预备语句或者结果集的信息

Synopsis

```
DESCRIBE [ OUTPUT ] _prepared_name_ USING [ SQL ] DESCRIPTOR _descriptor_name_
DESCRIBE [ OUTPUT ] _prepared_name_ INTO [ SQL ] DESCRIPTOR _descriptor_name_
DESCRIBE [ OUTPUT ] _prepared_name_ INTO _sqlda_name_
```

描述

`DESCRIBE` 将检索关于在预备语句中包含的结果列的元数据信息， 实际上没有读取行。

参数

`_prepared_name_`

预备语句名字可以是一个SQL标识符或者宿主变量。

`_descriptor_name_`

描述符名字大小写敏感。可以是SQL标识符或者宿主变量。

`_sqlda_name_`

SQLDA变量名字。

例子

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

兼容性

在SQL标准中声明 `DESCRIBE` 。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

Name

DISCONNECT -- 终止数据库连接

Synopsis

```
DISCONNECT _connection_name_
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

描述

`DISCONNECT` 关闭与数据库的连接（或者所有连接）。

参数

`_connection_name_`

通过 `CONNECT` 命令建立数据库连接名字。

`CURRENT`

关闭"当前"连接，它要么是最近打开的连接，或者是通过 `SET CONNECTION` 命令设置的连接。如果没有给出参数到 `DISCONNECT` 命令中，那么这也是缺省的。

`DEFAULT`

关闭缺省连接。

`ALL`

关闭所有打开连接。

例子

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* close con3          */
    EXEC SQL DISCONNECT DEFAULT;  /* close DEFAULT       */
    EXEC SQL DISCONNECT ALL;      /* close con2 and con1 */

    return 0;
}
```

兼容性

在SQL标准中声明 `DISCONNECT` 。

参见

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

Name

EXECUTE IMMEDIATE -- 动态准备和执行语句

Synopsis

```
EXECUTE IMMEDIATE _string_
```

描述

`EXECUTE IMMEDIATE` 立即准备并且执行动态声明的SQL语句，而 没有检索结果行。

参数

`_string_`

包含要执行的SQL语句文本C字符串或者宿主变量。

例子

这是一个使用 `EXECUTE IMMEDIATE` 和 命名 `command` 宿主变量执行 `INSERT` 语句的例子：

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')")
EXEC SQL EXECUTE IMMEDIATE :command;
```

兼容性

在SQL标准中声明 `EXECUTE IMMEDIATE` 。

GET DESCRIPTOR

Name

GET DESCRIPTOR -- 从SQL标识符区域获得信息

Synopsis

```
GET DESCRIPTOR _descriptor_name_ _:cvariable_ = _descriptor_header_item_ [, ... ]
GET DESCRIPTOR _descriptor_name_ VALUE _column_number_ _:cvariable_ = _descriptor_item_ [
```

描述

`GET DESCRIPTOR` 从SQL描述符区域检索关于查询结果集的信息， 并且将它存储到宿主变量中。在使用该命令将信息传递给宿主语言变量之前 通常使用 `FETCH` 或者 `SELECT` 填充标识符区域。

该命令有两种形式：第一个形式检索描述符"头部"项， 适用于作为整体的结果集。 第二个形式需要作为额外参数的列数检索关于特定列的信息。 例子是列名和实际列值。

参数

`_descriptor_name_`

描述符名字。

`_descriptor_header_item_`

一个标记识别检索的头部信息项。目前仅仅支持 `COUNT` 可以获取结果集中的列数。

`_column_number_`

关于被检索的列数信息。计数从1开始。

`_descriptor_item_`

一个标记识别检索列的信息项。参阅Section 33.7.1 获取可支持项的列数。

`_cvariable_`

宿主变量将接收从描述符区域检索的数据。

例子

检索结果集中列数的例子：

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

在第一列中检索数据长度的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

检索作为字符串第二列的数据主体的例子：

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

这是一个执行 `SELECT current_database();` 的整个程序的例子， 并且显示了列数，列数据长度和列数据：

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL ALLOCATE DESCRIPTOR d;

/*声明，打开游标，并且分配描述符给游标 */

    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

/*得到总列数*/

    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

/* 得到返回列的长度 */

    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

/*读取返回列作为字符串*/

    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

 /*关闭*/

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

当执行该例子的时候，结果看起来像这样：

```
d_count                 = 1
d_returned_octet_length = 6
d_data                  = testdb
```

兼容性

在SQL标准中指定 `GET DESCRIPTOR` 。

参见

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

Name

OPEN -- 打开一个动态游标

Synopsis

```
OPEN _cursor_name_
OPEN _cursor_name_ USING _value_ [, ... ]
OPEN _cursor_name_ USING SQL DESCRIPTOR _descriptor_name_
```

描述

`OPEN` 打开游标并且任意地绑定实际值到 游标声明中的占位符。游标必须预先使用 `DECLARE` 命令被声明。 `OPEN` 的执行导致查询在服务器上开始执行。

参数

`_cursor_name_`

要打开的游标名字。可以是SQL标识符或者宿主变量。

`_value_`

一个绑定到游标中占位符的值。可以是SQL常数， 宿主变量或者使用指示器的宿主变量。

`_descriptor_name_`

描述符名字包含绑定到游标中占位符的值。 可以是SQL标识符或者宿主变量。

例子

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

兼容性

在SQL标准中指定`OPEN`。

参见

DECLARE, CLOSE

PREPARE

Name

PREPARE -- 准备一个执行语句

Synopsis

```
PREPARE _name_ FROM _string_
```

描述

`PREPARE` 动态准备指定作为字符串执行的语句。 这不同于直接SQL语句PREPARE， 也可以作为嵌入程序使用。 EXECUTE命令用来执行 两种预备语句。

参数

`_prepared_name_`

预备查询标识符。

`_string_`

包含一个预备语句，SELECT，INSERT，UPDATE或者DELETE之一的文本C字符串或者宿主变量。

例子

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

兼容性

在SQL标准中指定 `PREPARE` 。

参见

EXECUTE

SET AUTOCOMMIT

Name

SET AUTOCOMMIT -- 设置当前会话自动提交操作

Synopsis

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

描述

`SET AUTOCOMMIT`设置当前数据库会话自动提交操作。 缺省，嵌入SQL程序不在自动提交模式，因此 当需要的时候，需要明确发布`COMMIT`。 该命令可以改变会话自动提交模式，则隐式提交每个单独语句。

兼容性

`SET AUTOCOMMIT`是PostgreSQL ECPG的一个扩展。

SET CONNECTION

Name

SET CONNECTION -- 选择一个数据库连接

Synopsis

```
SET CONNECTION [ TO | = ] _connection_name_
```

描述

`SET CONNECTION` 设置"当前"数据库连接， 这是所有命令使用的一个，除非重写。

参数

`_connection_name_`

通过 `CONNECT` 命令创建数据库连接名字。

`DEFAULT`

设置连接为缺省连接。

例子

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

兼容性

在SQL标准中指定 `SET CONNECTION` 。

参见

CONNECT, DISCONNECT

SET DESCRIPTOR

Name

SET DESCRIPTOR -- 设置SQL描述符区域信息

Synopsis

```
SET DESCRIPTOR _descriptor_name_ _descriptor_header_item_ = _value_ [, ... ]
SET DESCRIPTOR _descriptor_name_ VALUE _number_ _descriptor_item_ = _value_ [, ...]
```

描述

`SET DESCRIPTOR` 使用值填充SQL描述符区域。 描述符区域通常用于绑定预备查询执行中的参数。

该参数有两种形式：第一种形式适用于描述符"头部"， 它不受特定数据影响。第二中形式将值分配给特定数据，通过数字标识。

参数

`_descriptor_name_`

描述符名字。

`_descriptor_header_item_`

标记识别设置的头部信息项。目前仅仅支持 `COUNT` 设置描述符项数。

`_number_`

设置的描述符项数。计数从1开始。

`_descriptor_item_`

标记识别在描述符中的项信息。参阅Section 33.7.1 获取可支持项的列表。

`_value_`

存储到描述符项中的值。可以是SQl常量或者宿主变量。

例子

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

兼容性

在SQL标准中指定`SET DESCRIPTOR`。

参见

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

Name

TYPE -- 定义新的数据类型

Synopsis

```
TYPE _type_name_ IS _ctype_
```

描述

`TYPE` 命令定义一个新的C类型。 它相当于将 `typedef` 放到声明段中。

当运行带有 `-c` 选项的 `ecpg` 的时候， 仅仅标识该命令。

参数

`_type_name_`

新类型名字。它必须是一个有效的C类型名字。

`_ctype_`

C类型说明。

例子

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

这里是使用 `EXEC SQL TYPE` 的例子程序：

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

该程序的输出看起来像这样：

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

兼容性

`TYPE` 命令是PostgreSQL的扩展。

VAR

Name

VAR -- 定义一个变量

Synopsis

```
VAR _varname_ IS _ctype_
```

描述

`VAR` 命令将新的C数据类型分配给宿主变量。 宿主变量必须预先在声明段声明。

参数

`_varname_`

C变量名字。

`_ctype_`

C类型说明。

例子

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

兼容性

`VAR` 命令是PostgreSQL扩展。

WHENEVER

Name

WHENEVER -- 当SQL语句导致一个要发生的特定类条件时， 则指定要采取的行动

Synopsis

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } _action_
```

描述

定义在SQL执行结果中特殊情况下（没有发现行，SQL警告或者错误）调用的操作。

参数

参阅Section 33.8.1获取该参数的描述。

例子

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

一个典型应用是通过结果集处理循环的 `WHENEVER NOT FOUND BREAK` 的使用：

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

/*当到达结果集终端时，打破while循环*/

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

兼容性

在SQL标准中指定`WHENEVER`，但是大多数操作是PostgreSQL扩展。

33.15. Informix兼容模式

`ecpg` 可以在所谓的 *Informix*兼容模式下运行。 如果这种模式是活跃的，它试图表现得好像Informix E/SQL的Informix预编译器。 通常所说的这将允许你使用美元符号， 而不是 `EXEC SQL` 原始的引入嵌入式的SQL命令。

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **Note:** 在 `$` 之间不能有任何空格和下面的预处理指令， 即 `include` ， `define` ， `ifdef` 等等。否则，预处理器将解析令牌作为宿主变量。

有两种兼容模式： `INFORMIX` , `INFORMIX_SE` 。

当使用这些兼容模式连接程序时，记得要链接 `libcompat` 前内置于ECPG中。

除了前面所说的语法块， Informix兼容模式为输入输出和数据转换功能， 以及从E/SQL到ECPG嵌入式SQL语句提供一些函数。

Informix兼容模式紧密联系ECPG的pgtypeslib库。 pgtypeslib在C主机程序映射SQL数据类型到数据类型，并且 Informix 兼容模式的大部分附加函数允许你在那些C主机程序类型上进行操作。 但值得注意的是，兼容性的程度是有限的。它并不试图复制Informix的操作；它允许你做或多或少相同的操作，使你具有相同的名称和相同的基本行为功能，但如果你现在使用Informix，它没有下拉式功能表位置。 此外，一些数据类型是不同的。比如，PostgreSQL的datetime和 区间类型不知道范围比如 `YEAR TO MINUTE` ， 因此你在ECPG的这两种类型中不会找到支持。

33.15.1. 附加类型

在不使用 `typedef` 的Informix模式中 现在支持为存储正确修剪字符串数据 Informix特殊"字符串"伪类型。 实际上，在Informix模式中，ECPG拒绝包含 `typedef sometype string;` 的过程源文件。

```
EXEC SQL BEGIN DECLARE SECTION;

string userid; /* 这个变量将包含修剪的数据 */

EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

33.15.2. 附加的/失踪的嵌入的SQL语句

`CLOSE DATABASE`

这个语句关闭当前连接。事实上，这是ECPG的 `DISCONNECT CURRENT` 的同义词：

```
$CLOSE DATABASE;                /* 关闭当前连接 */
EXEC SQL CLOSE DATABASE;
```

`FREE cursor_name`

由于ECPG与Informix的ESQL/C如何运行的不同（即这个步骤是纯粹的语法转换并且依赖于底层运行时的库）在ECPG中没有 `FREE cursor_name` 声明。 这是因为在ECPG中， `DECLARE CURSOR` 不转化为使用游标名称调用运行时库的一个函数。 这意味着，在ECPG中运行时库中SQL游标没有运行时的记录，只有在PostgreSQL服务器上。

`FREE statement_name`

`FREE statement_name` 是 `DEALLOCATE PREPARE statement_name` 的同义词。

33.15.3. Informix兼容SQLDA描述符区域

Informix兼容模式支持一个 比Section 33.7.2描述中的不同的结构。见下文：

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

全局属性是：

`sqld`

在 `SQLDA` 描述符中的字段的数目。

`sqlvar`

每个字段属性的指针。

`desc_name`

未使用的，填充零字节。

`desc_occ`

分配的结构大小。

`desc_next`

如果结果集中包含1个以上的记录，指向下一个SQLDA结构的指针。

`reserved`

未使用指针，包含空。保持Informix兼容。

下面每个字段属性，它们被存储在 `sqlvar` 数组中：

`sqltype`

字段类型。常量在 `sqltypes.h` 中。

`sqllen`

字段数据的长度。

`sqldata`

指向字段数据的指针。指针是 `char *` 类型， 通过它指出的数据是二进制格式。例子:

```
int intval;

switch (sqldata-&gt;sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata-&gt;sqlvar[i].sqldata;
        break;
  ...
}
```

`sqlind`

指向空指针。如果通过DESCRIBE或取回后，那么它总是一个有效的指针。 如果使用 `EXECUTE ... USING sqlda;` 作为输入，然后空指针值意味着该字段的值为非空。否则，一个有效的指针 和 `sqlitype` 必须正确设置。例子:

```
if (*(int2 *)sqldata-&gt;sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

`sqlname`

字段名称。0终止字符串。

`sqlformat`

保留在Informix中， `PQfformat()` 的值为字段的值。

`sqlitype`

空指针的数据类型。当从服务器返回数据时，它总是SQLSMINT。 当 `SQLDA` 用于参数化查询时， 数据是根据设定的类型来处理的。

`sqlilen`

空指针数据的长度。

`sqlxid`

字段的扩展类型，结果 `PQftype()` 。

`sqltypename``sqltypelen` `sqlownerlen` `sqlsourcetype` `sqlownername` `sqlsourceid` `sqlflags` `sqlreserved`

未使用。

`sqlilongdata`

如果 `sqllen` 大于32KB。它等于 `sqldata` 。

例子：

```
EXEC SQL INCLUDE sqlda.h;

sqlda_t        *sqlda; /* 这个不需要在嵌入DECLARE SECTION的下面 */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;

    while (1)
    {
      EXEC SQL FETCH mycursor USING sqlda;
    }

    EXEC SQL CLOSE mycursor;

free(sqlda); /* 主结构被free(), sqlda和sqlda->sqlvar在分配区域中*/
```

获取更多信息，参见 `sqlda.h` 头文件和 `src/interfaces/ecpg/test/compat_informix/sqlda.pgc` 回归测试。

33.15.4. 附加函数

`decadd`

增加2个decimal类型值。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

这个函数接受一个指向类型decimal(`arg1`)的第一个操作数的指针， 类型decimal (`arg2`)的第二个操作数的指针和包含sum (`sum`)类型decimal值的指针。成功时，函数返回0。 以免溢出返回 `ECPG_INFORMIX_NUM_OVERFLOW` 和在下溢的情况下返回 `ECPG_INFORMIX_NUM_UNDERFLOW` 。其他错误返回-1， `errno` 设置为pgtypeslib的各自的 `errno` 数。

`deccmp`

比较decimal类型的2个变量。

```
int deccmp(decimal *arg1, decimal *arg2);
```

这个函数接受一个指向第一个decimal值(`arg1`)的指针， 一个指向第二个decimal值(`arg2`)的指针并返回 一个表示这是更大值的整数值。

- 如果 `arg1` 指向比 `var2` 指向的值更大，则为1。
- 如果 `arg1` 指向比 `arg2` 指向的值更小，则为-1。
- 如果 `arg1` 指向和 `arg2` 指向的值相等，则为0。

`deccopy`

复制一个decimal值。

```
void deccopy(decimal *src, decimal *target);
```

这个函数接受一个指向复制为第一个参数(`src`) 的decimal值的指针和 一个指向decimal(`target`)的目标结构作为第二个参数的指针。

`deccvasc`

将一个值从ASCII表示转换为decimal类型。

```
int deccvasc(char *cp, int len, decimal *np);
```

这个函数接受一个指向字符串的指针， 这个字符串包含转换成和其长度一样为 `len` 的(`cp`)数字的字符串表示形式。 `np` 是一个指向decimal值的指针，节省了运算的结果。

有效格式比如： `-2` ， `.794` ， `+3.44` ， `592.49E07` 或者 `-32.84e-4` 。

函数成功时返回0。如果发生溢出或下溢， 返回 `ECPG_INFORMIX_NUM_OVERFLOW` 或者 `ECPG_INFORMIX_NUM_UNDERFLOW` 。如果ASCII表示不能被解析， 返回 `ECPG_INFORMIX_BAD_NUMERIC` ， 如果解析指数发生此问题时返回 `ECPG_INFORMIX_BAD_EXPONENT` 。

`deccvdbl`

转换double类型的值到一个decimal类型的值。

```
int deccvdbl(double dbl, decimal *np);
```

函数接收一个被作为第一个参数(`dbl`)转换的double类型的变量。 作为第二个参数(`np`)， 函数接收一个指向保持操作结果的decimal变量的指针。

成功时函数返回0，如果转换失败则返回一个负数。

`deccvint`

转换一个int类型的值到一个decimal类型的值。

```
int deccvint(int in, decimal *np);
```

函数接收应作为第一个参数(`in`)转换的int类型的变量。作为第二个参数(`np`)，函数接收一个指向保持操作结果的decimal变量的指针。

成功时函数返回0，如果转换失败则返回一个负数。

`deccvlong`

转换一个long类型的值到一个decimal类型的值。

```
int deccvlong(long lng, decimal *np);
```

函数接收应作为第一个参数(`lng`)转换的long类型的变量。 作为第二个参数(`np`)，函数接收一个指向保持操作 结果的decimal变量的指针。

成功时函数返回0，如果转换失败则返回一个负数。

`decdiv`

decimal类型的两个变量相除。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

函数接收一个指向第一个(`n1`)和第二个(`n2`)操作数变量的指针并且计算 `n1` / `n2` 。 `result` 是指向保持操作结果的变量的指针。

成功则返回0，如果除法运算失败则返回一个负数。如果产生溢出或者下溢，函数各自返回 `ECPG_INFORMIX_NUM_OVERFLOW` 或者 `ECPG_INFORMIX_NUM_UNDERFLOW` 。 如果观察到尝试除以0，函数返回 `ECPG_INFORMIX_DIVIDE_ZERO` 。

`decmul`

两个decimal值相乘。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

函数接收指向第一个(`n1`)和第二个(`n2`)操作数变量的指针并且计算 `n1` * `n2` 。 `result` 是指向保持操作结果的变量的指针。

成功则返回0，如果乘法运算失败则返回一个负数。如果发生溢出或者下溢，函数各自返回 `ECPG_INFORMIX_NUM_OVERFLOW` 或者 `ECPG_INFORMIX_NUM_UNDERFLOW` 。

`decsub`

一个decimal值与另一个值相减。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

函数接收指向第一个(`n1`)和第二个(`n2`)操作数变量的指针并且计算 `n1` - `n2` 。 `result` 是指向保持操作结果的变量的指针。

成功则返回0，如果减法运算失败则返回一个负数。如果发生溢出或者下溢，函数各自返回 `ECPG_INFORMIX_NUM_OVERFLOW` 或者 `ECPG_INFORMIX_NUM_UNDERFLOW` 。

`dectoasc`

转换一个decimal类型变量到C char*字符串中ASCII表示。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

函数接收一个指向decimal(`np`)类型文本表示的变量的指针。 `cp` 是保存操作结果的缓冲区。参数 `right` 指明小数点右边有多少位数字应该包含在输出中。结果将四舍五入小数数字位数。设置 `right` 到-1表明所有可用的小数位数应包含在输出中。如果由 `len` 指明的输出缓冲区的长度不足以保持包含尾随NUL字符的文本表示，仅仅在结果中存储一个单一的 `*` 字符并且返回-1。

如果缓冲区 `cp` 太小，函数则返回-1，如果内存耗尽， 则返回 `ECPG_INFORMIX_OUT_OF_MEMORY` 。

`dectodbl`

转换一个decimal类型的变量到一个double类型。

```
int dectodbl(decimal *np, double *dblp);
```

函数接受一个指向decimal值转换（ `np` ） 和 一个指向应保持操作结果(`dblp`)的double变量的指针。

成功则返回0，如果转换失败则返回一个负数。

`dectoint`

转化一个decimal类型到一个integer类型的变量。

```
int dectoint(decimal *np, int *ip);
```

函数接受一个指向decimal值转换(`np`)和 指向应保持操作结果（ `ip` ） 的integer变量的指针。

成功则返回0，如果转换失败则返回一个负数。如果发生溢出， 则返回 `ECPG_INFORMIX_NUM_OVERFLOW` 。

注意ECPG应用不同于Informix应用。 当ECPG应用中的限制取决于(`-INT_MAX .. INT_MAX`)的结构，Informix 限制integer的范围从-32767到32767。

`dectolong`

转换一个decimal类型的变量到一个long integer类型。

```
int dectolong(decimal *np, long *lngp);
```

函数接受一个decimal值转换(`np`)和 一个应保持操作结果(`lngp`)的long变量的指针。

成功则返回0，如果转换失败则返回一个负数。 如果发生溢出，则返回 `ECPG_INFORMIX_NUM_OVERFLOW` 。

注意ECPG应用不同于Informix应用。当ECPG应用中的限制取决于(`-LONG_MAX ..LONG_MAX`)的结构，Informix限制long integer的范围从-2,147,483,647到2,147,483,647。

`rdatestr`

转换一个date类型到C char*字符串。

```
int rdatestr(date d, char *str);
```

这个函数接收2个参数，第一个是日期转换（ `d` 和第二个参数是一个指向目标字符串的指针。输出格式总是 `yyyy-mm-dd` ，因此你需要为字符串至少分配11个字节（包含0字节终止符）。

成功函数则返回0，出错时则返回一个负数。

注意ECPG应用不同于Informix应用。 Informix中格式受到环境变量设置的影响。 然而在ECPG中，你不能改变输出格式。

`rstrdate`

解析日期的文本表示。

```
int rstrdate(char *str, date *d);
```

这个函数接受日期转换(`str`)的文本表示形式和指向类型date(`d`)的变量的指针。这个函数不允许你指明格式掩码。它使用Informix缺省格式掩码 `mm/dd/yyyy` 。实质上，它通过 `rdefmtdate` 实现的。因此， `rstrdate` 不是很快，如果你有选择了，你应该选择 `rdefmtdate` ，它允许你明确指定掩码格式。

函数返回和 `rdefmtdate` 一样的值。

`rtoday`

获取当前日期。

```
void rtoday(date *d);
```

函数接受一个设置当前日期的日期变量(`d`)的指针。

实质上这个函数使用 *PGTYPESdate_today* 函数。

`rjulmdy`

从date类型的变量中提取一天，一个月，一年中的值。

```
int rjulmdy(date d, short mdy[3]);
```

这个函数接受日期 `d` 和一个指向3个短整型数值 `mdy` 数组的指针，变量名表明了相继顺序： `mdy[0]` 的设置包含了月数， `mdy[1]` 的设置是一天的值， `mdy[2]` 包含年的值。

函数此时总是返回0。

实质上函数使用 *PGTYPESdate_julmdy* 函数。

`rdefmtdate`

使用格式掩码转换字符串到一个date类型的值。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

函数接受一个保持操作(`d`)结果日期值的指针。 格式掩码用于解析日期(`fmt`)和包含date(`str`)文本表示形式的C char*字符串。文本表示形式期望匹配格式掩码。然而你不需要有一个1:1字符串映射格式掩码。它仅分析相继顺序并且查找 `yy` 或者 `yyyy` 表明年的位置， `mm` 表示月份 和 `dd` 表示一天的位置。

函数返回下列值:

- 0 - 函数成功终止。
- `ECPG_INFORMIX_ENOSHORTDATE` - 在日，月和年之间日期不包含定界符。 在这种情况下输入字符串必须确切地6或8个字节但是不是这样的。

- `ECPG_INFORMIX_ENOTDMY` - 格式字符串不能正确显示年，月，日的相继顺序。
- `ECPG_INFORMIX_BAD_DAY` - 输入字符串不包含一个有效的天数。
- `ECPG_INFORMIX_BAD_MONTH` - 输入字符串不包含一个有效的月份。
- `ECPG_INFORMIX_BAD_YEAR` - 输入字符串不包含一个有效的年份。

实质上这个函数使用 *PGTYPESdate_defmt_asc* 函数来实现。 请参阅这儿的实例输入的表。

`rfmtdate`

转换一个date类型变量到它的一个格式掩码的文本表示。

```
int rfmtdate(date d, char *fmt, char *str);
```

这个函数接收一个日期转换(`d`)， 格式掩码(`fmt`)和将保持日期(`str`)的文本表示形式 的字符串。

成功返回0，如果产生错误则返回一个负值。

实质上这个函数 使用 *PGTYPESdate_fmt_asc* 函数。请参阅这儿的例子。

`rmdyjul`

从指明日期中的年，月，日的3维短整型数组中创建一个日期值。

```
int rmdyjul(short mdy[3], date *d);
```

这个函数接收一个3维短整型(`mdy`)数组 和一个指向应保持操作结果的日期类型变量的指针。

当前这个函数总是返回0。

实质上这个函数使用 *PGTYPESdate_mdyjul* 来实现。

`rdayofweek`

返回一个数字表示的一个日期值中某一周的某一天。

```
int rdayofweek(date d);
```

函数接收日期变量 `d` 作为其唯一的参数并返回一个整数， 表示这一天是星期几。

- 0 - 星期日
- 1 - 星期一
- 2 - 星期二
- 3 - 星期三

- 4 - 星期四
- 5 - 星期五
- 6 - 星期六

实质上这个函数使用 *PGTYPESdate_dayofweek* 来实现。

`dtcurrent`

检索当前的timestamp。

```
void dtcurrent(timestamp *ts);
```

这个函数检索当前timestamp并且保存 它到 `ts` 指向的timestamp变量中。

`dtcvasc`

解析一个文本表示的timestamp到一个timestamp变量。

```
int dtcvasc(char *str, timestamp *ts);
```

这个函数接收解析字符串（ `str` ） 和一个指向应保持操作结果（ `ts` ） 的timestamp变量的指针。

成功时函数返回0，发生错误的时候返回负数。

实质上这个函数使用 *PGTYPEStimestamp_from_asc* 函数。 请参见这儿的实例输入的表。

`dtcvfmtasc`

使用格式掩码解析文本表示的timestamp为timestamp变量。

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

这个函数接收解析(`inbuf`)字符串， 使用(`fmtstr`)格式掩码， 以及一个指向保持操作(`dtvalue`)结果的timestamp变量的指针。

这个函数 通过 *PGTYPEStimestamp_defmt_asc* 函数来实现。参见文档中使用的格式说明符列表。

成功时函数返回0，发生错误的时候则返回一个负数。

`dtsub`

一个timestamp与另一个相减，并且返回一个区间类型变量。

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

该函数将时间戳变量相减， 其中 `ts2` 指向 `ts1` 指向的时间戳变量， 并将结果存储在 `iv` 指向的区间变量中。

成功时函数返回0，如果发生错误则返回一个负数。

`dttoasc`

转换一个timestamp变量到C char*字符串。

```
int dttoasc(timestamp *ts, char *output);
```

这个函数接收一个指向timestamp变量转换(`ts` ） 的指针和保持操作 `output` 结果的字符串。 根据SQL标准，转换 `ts` 到它的一个文本表示形式， 即 `YYYY-MM-DD HH:MM:SS` 。

成功函数返回0，如果产生错误则返回负数。

`dttofmtasc`

使用格式掩码将timestamp变量转换成C char*。

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

这个函数接收一个指向作为第一个参数(`ts`)转换的timestamp的指针，以及指向输出缓存（ `output` ）的指针， 已经分配给输出缓存(`str_len`)的 最大长度和用于转换（ `fmtstr` ）的格式掩码。

成功时函数返回0，如果产生错误则返回一个负数。

实质上，这个函数 使用 *`PGTYPEStimestamp_fmt_asc`* 函数。 请参阅有关格式掩码说明的使用获取更多信息。

`intoasc`

将区间变量转换成C char*字符串。

```
int intoasc(interval *i, char *str);
```

这个函数接收一个指向区间 变量转换(`i`)的指针和 保持操作(`str`)结果的字符串。 它依照SQL标准将 `i` 转换成文本表示形式， 即 `YYYY-MM-DD HH:MM:SS` 。

成功时函数返回0，如果产生错误则返回负数。

`rfmtlong`

将长整型数值转换成使用格式掩码的文本表示形式。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

这个函数接收长值 `lng_val`， 格式掩码 `fmt` 和指向输出缓存 `outbuf` 的指针。 它依照格式掩码将长值转换成文本表示形式。

格式掩码由下面格式指定字符组成。

- `*` (星号) – 如果这个位置是空白的，否则，用星号填充。
- `&` (&符号) -如果这个位置是空白的，否则，用零填充。
- `#` - 将前导零转换成空格。
- `<` -字符串中左对齐数字。
- `,` (逗号)–将四个或更多数字分成由逗号分隔的三个数字一组。
- `.` (句号) – 这个字符将从小数部分分离出数的整数部分。
- `-` (减号) – 如果数是负值，则出现负号。
- `+` (加号) -如果数是正值，则出现加号。
- `(` -这将替代负数前面的减号。减号将不会出现。
- `)` -这个字符替代减号，并且在负数后面输出。
- `$` - 货币符号。

`rupshift`

将字符串转换为大写。

```
void rupshift(char *str);
```

这个函数接收一个指向字符串的指针，并且将每个小写字符转换成大写字符。

`byleng`

返回没有计算空格的字符串中字符的数量。

```
int byleng(char *str, int len);
```

这个函数期望固定长度字符串作为它第一个参数(`str`)， 并且其长度作为第二个参数(`len`)。 它返回重要字符数，它是没有空格的字符串的长度。

`ldchar`

复制一个固定长度字符串到空终止符字符串中。

```
void ldchar(char *src, int len, char *dest);
```

函数接收固定长度的字符串复制（`src`）， 其长度（`len`）和一个指向目标内存（`dest`）的指针。 注意，你需要为`dest`指向的字符串至少储备`len+1`字节。 函数复制最多`len`字节到新的位置（至少如果源字符串有尾随空格） 和添加空终止符。

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

这个函数存在但此刻不能实现。

`rtypalign`

```
int rtypalign(int offset, int type);
```

这个函数存在但此刻不能实现。

`rtypmsize`

```
int rtypmsize(int type, int len);
```

这个函数存在但此刻不能实现。

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

这个函数存在但此刻不能实现。

`rsetnull`

设置一个变量为空。

```
int rsetnull(int t, char *ptr);
```

这个函数接收一个表明变量类型的整数和一个指向映射到C char*指针的变量自身的指针。

存在下面类型:

- `CCHARTYPE` - 为了类型`char`或者`char*`的变量
- `CSHORTTYPE` - 为了类型`short int`的变量
- `CINTTYPE` -为了类型为`int`的变量
- `CBOOLTYPE` -为了类型`boolean`的变量
- `CFLOATTYPE` - 为了类型`float`的变量

- `CLONGTYPE` - 为了类型 `long` 的变量
- `CDOUBLETYPE` - 为了类型 `double` 的变量
- `CDECIMALTYPE` - 为了类型 `decimal` 的变量
- `CDATETYPE` - 为了类型 `date` 的变量
- `CDTIMETYPE` - 为了类型 `timestamp` 的变量

下面是调用这个函数的一个例子：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

如果变量是空，测试：

```
int risnull(int t, char *ptr);
```

这个函数接收测试（ `t` ）变量的类型以及 一个指向这个变量（ `ptr` ）的指针。 注意后者需要映射到一个char*。 参见函数 *`rsetnull`* 获取可能变量类型的列表。

这儿是如何使用这个函数的一个例子：

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

33.15.5. 附加常量

注意，这里所有的常量描述错误，他们都被定义为代表负数。 不同常量的描述中，你还可以发现，目前的应用中表示常量的值。 然而你不应该依赖于数量。但是你可以依靠它们被定义为表示负数的事实。

`ECPG_INFORMIX_NUM_OVERFLOW`

如果在计算中发生溢出，函数返回这个值。 实质上它被定义为-1200（Informix定义）。

`ECPG_INFORMIX_NUM_UNDERFLOW`

如果在计算中发生下溢，函数返回这个值。 实质上它被定义为-1201(Informix定义)。

`ECPG_INFORMIX_DIVIDE_ZERO`

如果观察到尝试除以零，函数返回这个值。实质上它被定义为 -1202（Informix定义）。

`ECPG_INFORMIX_BAD_YEAR`

如果当解析一个日期时，发现某年的一个错误值，函数返回这个值。实质上它被定义为-1204(Informix定义)。

`ECPG_INFORMIX_BAD_MONTH`

如果当解析一个日期时，发现某月的一个错误值，函数返回这个值。实质上它被定义为-1205(Informix定义)。

`ECPG_INFORMIX_BAD_DAY`

如果解析一个日期的时候，发现某天的错误值，函数返回这个值，实质上它被定义为-1206(Informix定义)。

`ECPG_INFORMIX_ENOSHORTDATE`

如果解析程序需要一个短日期表示形式而没有获得正确长度的日期字符串，函数则返回这个数值。 实质上它被定义为-1209(Informix定义)。

`ECPG_INFORMIX_DATE_CONVERT`

如果在日期格式之间发生错误，那么函数返回该值。 实质上它被定义为-1210(Informix定义)。

`ECPG_INFORMIX_OUT_OF_MEMORY`

如果在操作期间内存耗尽，那么函数返回该值。实质上 它被定义为-1211（Informix定义）。

`ECPG_INFORMIX_ENOTDMY`

如果解析程序应该得到一个掩码格式 （如 `mmddyy` ）但不是所有的字段都被正确地列出来，函数返回该值。 实质上它被定义为-1212(Informix定义)。

`ECPG_INFORMIX_BAD_NUMERIC`

如果一个解析程序无法解析为数字值的文本表示， 因为它包含错误，或者如果程序不能完成涉及数值变量的计算， 因为至少这些数值变量之一是无效的。那么该函数返回这个值。 实质上它被定义为-1213(Informix定义)。

`ECPG_INFORMIX_BAD_EXPONENT`

如果解析程序无法解析指数，那么该函数返回此值。 实质地被定义为-1216 （Informix定义）。

`ECPG_INFORMIX_BAD_DATE`

如果解析程序不能解析日期，那么该函数返回此值。 实质上它被定义为-1218（Informix定义）。

`ECPG_INFORMIX_EXTRA_CHARS`

如果不能解析的多余字符被传递给解析程序，那么该函数返回这个值。 实质上它被定义为-1264（Informix定义）。

33.16. 内部

这一节解释ECPG在内部是如何运转的。 这些信息有时候可以帮助用户理解如何使用ECPG。

`ecpg` 写到输出里的头四行是固定的行。两行是注释， 另外两行是与库接口的必要行。 然后预处理器读取文件并且写输出流。 通常它只是把所有东西都回显到输出中去。

如果它看到一个 `EXEC SQL` 语句， 它就变换并且修改它。命令以 `EXEC SQL` 开头，以 `;` 结尾。所有在中间的东西都被当 作一个SQL语句并且进行变量代换的解析。

如果一个符号以一个冒号（ `:` ）开头，则发生变量代换。 在 `EXEC SQL DECLARE` 段里预先声明的变量中找出该名字的变量。

库里面最重要的函数是 `ECPGdo` ，它负责执行大多数命令。 它接受变量的参数个数。 这些参数的个数可能很容易达到50个或者更多， 我们希望在任何平台上这都不是问题。

参数是：

行号

这是原始行的行号；只是在错误信息中使用。

字符串

这是要发出的SQL命令。 它被输入变量修改，也就是说，在编译时未知的， 但是需要输入到命令中的变量。 此处变量应该包含字符串 `?` 。

输入变量

每个输入变量都导致十个参数的生成。（见下文）

`ECPGt_EOIT`

一个 `enum` 告诉没有更多输入变量了。

输出变量

每个输出变量导致十个参数的创建。（见下文） 这些变量被这些函数填充。

`ECPGt_EORT`

一个指出没有更多变量的 `enum` 。

对于每个属于SQL命令一部分的变量， 函数可以得到十个参数：

1. 作为特殊符号的类型。
2. 一个指向其数值的指针，或者一个指向指针的指针。

3. 如果变量大小是 `char` 或者 `varchar` 。
4. 数组中元素的个数（用于抓取数组）。
5. 指向数组中下一个元素的偏移量（用于抓取数组）。
6. 作为一种特殊符号的指示器变量的类型。
7. 一个指向指示器变量的指针。
8. 0
9. 指示器数组中的元素个数（用于抓取数组）。
10. 指向指示器数组的下一个元素的偏移量（用于抓取数组）。

请注意，不是所有SQL命令都这么被对待。 比如，一个像下面这样的打开游标的语句。

```
EXEC SQL OPEN _cursor_;
```

不会被拷贝到输出中。 而是在 `OPEN` 命令的位置使用游标的 `DECLARE` 命令， 因为它同样也打开游标。

下面是一个完整的例子，描述了文件 `foo.pgc` 的预处理后的输出（细节可能随着每个不同的预处理器版本而变化）：

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

被翻译成：

```
/* 通过ecpg (2.6.0)处理，通过预处理器添加2个include文件*/

#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql开始声明段 */

#line 1 "foo.pgc"

 int index;
 int result;

/* exec sql结束声明段 */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(这里的凹进是为了增强可读性加的，可不是预处理器能干的事情。)

Chapter 34. 信息模式

Table of Contents

信息模式由一组视图组成，它们包含有关当前数据库里定义的对象的信息。 信息模式是 SQL 标准里定义的，因此可以认为是可以移植的，并且是相对稳定的— 和系统表不一样，系统表是PostgreSQL特有的， 是在实现的基础上进行建模的。但信息模式视图不包含有关 PostgreSQL 特有的特性的信息；你可以查询系统表或者其它PostgreSQL 特定的视图查询它。

> **Note:** 当查询数据库约束信息时，一个标准兼容的查询有可能返回一到多行。 这是因为在一个模式中SQL标准查询约束名是唯一的，但是PostgreSQL 并不强制这个约束。PostgreSQL 自动产生约束名避免在同一个模式中重复，但是用户可以指定重复的名字。
>
> 当查询信息模式视图（如 `check_constraint_routine_usage` , `check_constraints` , `domain_constraints` ,和 `referential_constraints` ）时会出现这样的问题。 一些其他视图有相似的问题，但是包含表明以帮助辨别重复的行， 例如 `constraint_column_usage` , `constraint_table_usage` , `table_constraints` 。

34.1. 关于这个模式

信息模式本身是一个叫`information_schema`的模式。 这个模式自动存在于所有数据库中。这个模式的所有者是数据库集群中的最初的数据库用户， 并且这个用户天然就拥有这个模式上的所有权限，包括删除它的权限 （不过这么干省下来的空间小的可怜）。

缺省的时候，信息模式不在模式搜索路径中，因此，你需要用全称来访问里面的所有对象。因为信息模式里的一些对象的名字是可能在用户应用中出现的普通名字，所以， 如果你想把信息模式放在路径中的话，那你一定要小心。

34.2. 数据类型

信息模式视图的字段使用的是特殊的数据类型，它们是在信息模式里定义的。 这些都是在内置类型上定义的简单的域。你不应当在信息模式之外的地方使用这些类型工作， 但是如果你的应用从信息模式中选取了数据，那么它必须面对它们。

这些类型是：

`cardinal_number`

非负整数。

`character_data`

一个字符串（没有声明最大长度）。

`sql_identifier`

一个字符串。这个类型用于SQL标识符， 类型 `character_data` 用于任何其它类型的文本数据。

`time_stamp`

一个在类型 `timestamp with time zone` 上的域。

`yes_or_no`

一个字符串域要么包含 `YES` 要么包含 `NO` 。 这在信息模式中用来表示Boolean (true/false)数据。 （在 `boolean` 类型被添加到SQL标准之前信息模式被发明出来， 为保持信息模式向前兼容这个约定是必须的。）

信息模式里的每个字段都有这五种类型之一。

34.3. `information_schema_catalog_name`

`information_schema_catalog_name`是一个总是包含一行一列的表， 里面包含当前数据库的名字（用SQL术语来说，是当前目录）。

Table 34-1. `information_schema_catalog_name` 字段

名字	数据类型	描述
`catalog_name`	`sql_identifier`	包含这个信息模式的数据库名字

34.4. `administrable_role_authorizations`

视图 `administrable_role_authorizations` 标识当前用户有管理员选项的所有角色。

Table 34-2. `administrable_role_authorizations` 字段

名字	数据类型	描述
`grantee`	`sql_identifier`	赋予角色成员的角色名称（可以是当前用户，或者一个在角色成员关系中嵌套的不同角色）
`role_name`	`sql_identifier`	角色名称
`is_grantable`	`yes_or_no`	总是 `YES`

34.5. `applicable_roles`

视图 `applicable_roles` 标识当前用户所属的所有组。 这意味着有一些角色束缚限制由当前用户授予问题中的角色。当前用户本身也是一个可用的角色。 这个适合的可用角色组通常被用做许可检查。

Table 34-3. `applicable_roles` 字段

名字	数据类型	描述
`grantee`	`sql_identifier`	赋予角色成员的角色名称（可以是当前用户，或者一个在角色成员关系嵌套中的不同的角色成员）
`role_name`	`sql_identifier`	角色名
`is_grantable`	`yes_or_no`	如果授权者有管理员权限的话为 `YES` , 否则为 `NO`

34.6. `attributes`

视图 `attributes` 包含有关在数据库中定义的复合数据类型的属性信息。 （注意，视图不会提供表字段的信息，在PostgreSQL环境中有时被叫做属性。） 只有当前用户有权限时才显示这些属性（通过成为它的所有者或者在这个类型上有某些权限）。

Table 34-4. `attributes` 字段

名字	数据类型	描述
`udt_catalog`	`sql_identifier`	包含数据类型的数据库名称（总是在当前数据库中）
`udt_schema`	`sql_identifier`	包含数据类型的模式名称
`udt_name`	`sql_identifier`	数据类型名称
`attribute_name`	`sql_identifier`	属性名
`ordinal_position`	`cardinal_number`	数据类型中的属性的顺序位置（从1开始计数）
`attribute_default`	`character_data`	属性的默认表达式
`is_nullable`	`yes_or_no`	如果属性可能为空则为 `YES`，如果不为空则为 `NO`。
`data_type`	`character_data`	如果它是内建类型，那么是属性的数据类型，或者如果它是一些数组， 那么是 `ARRAY` （在这种情况下， 查看视图 `element_types` ）， 其它情况是 `USER-DEFINED` （在这种情况下，在 `attribute_udt_name` 和相关字段中定义类型）。
`character_maximum_length`	`cardinal_number`	如果 `data_type` 标识一个字符或比特流类型，那么是声明的最大长度； 对于所有其它的数据类型或如果没有声明最大长度都用null。
`character_octet_length`	`cardinal_number`	如果 `data_type` 识别一个字符类型， 那么最大可能长度在一个数据的字节（bytes）中；对于所有其它的数据类型为null。最大的字节长度取决于声明的字符最大长度（参考上文）和服务器的编码。
`character_set_catalog`	`sql_identifier`	在PostgreSQL中的不适用特性

character_set_schema	sql_identifier	在PostgreSQL中的不适用特性
character_set_name	sql_identifier	在PostgreSQL中的不适用特性
collation_catalog	sql_identifier	包含属性排序规则的数据库名称（总是当前数据库）， 缺省或属性的数据类型不可排序时为null。
collation_schema	sql_identifier	包含属性排序规则的模式名，缺省或属性的数据类型不可排序时为null。
collation_name	sql_identifier	属性的排序规则的名称，缺省或属性的数据类型不可排序时为null。
numeric_precision	cardinal_number	如果 data_type 标识一个数字类型，那么该字段 （隐含地或者公开地）包含属性类型的精度。该精度表明了有效数字的位数。 它可以用在十进制（base 10）或者二进制（base 2）中，按照说明在字段 numeric_precision_radix 中。对于所有其它的数据类型， 该字段为null。
numeric_precision_radix	cardinal_number	如果 data_type 标识一个数字类型， 该字段表明基于字段 numeric_precision 和字段 numeric_scale 中的值。 值要么是2要么是10. 对于所有其它的数据类型， 字段是null。
numeric_scale	cardinal_number	如果 data_type 标识一个精确的数字类型，那么该字段 （公开地或隐含地）包含属性类型的数值范围。该数值范围表明了小数点右边有效数字位数。它可以用在十进制(base 10)或者二进制(base 2)中，详细说明在字段 numeric_precision_radix 中。对于所有其它的数据类型， 该字段为null。
datetime_precision	cardinal_number	如果 data_type 标识一个日期，时间，时间戳，或者间隔类型， 该字段（公开地或隐含地）包含属性类型的精确到小数部分的秒，就是说， 秒的值小数位数保持到小数点后面。对于所有其它的数据类型， 该字段是null。

`interval_type`	`character_data`	如果 `data_type` 标识一个间隔类型， 该字段包含为这个属性包含的字段的时间间隔说明，例如， `YEAR TO MONTH` ， `DAY TO SECOND` 等等。 如果没有声明字段限制（也就是， 间隔接受所有字段）， 和对所有其他数据类型， 该字段都是null。
`interval_precision`	`cardinal_number`	在PostgreSQL中的不适用特性 （参阅 `datetime_precision` 获取间隔类型属性的小数秒精度）
`attribute_udt_catalog`	`sql_identifier`	属性数据类型被指定的数据库名（总是在当前数据库中）
`attribute_udt_schema`	`sql_identifier`	属性数据类型被指定的模式名称
`attribute_udt_name`	`sql_identifier`	属性数据类型名称
`scope_catalog`	`sql_identifier`	在PostgreSQL中的不适用特性
`scope_schema`	`sql_identifier`	在PostgreSQL中的不适用特性
`scope_name`	`sql_identifier`	在PostgreSQL中的不适用特性
`maximum_cardinality`	`cardinal_number`	总是null，因为在PostgreSQL中数组的最大基数总是不受限
`dtd_identifier`	`sql_identifier`	字段的数据类型描述符的一个标示符，在表格的数据类型描述符中是唯一的。 这个对于加入其它的这样的标示符的实例是主要有用的。 （标示符的特定格式没有指定且也不保证在以后的版本中不会改变）
`is_derived_reference_attribute`	`yes_or_no`	在PostgreSQL中的不适用特性

也可以在Section 34.16中查阅，一个类似的结构视图，一些字段的进一步详细信息。

34.7. `character_sets`

视图`character_sets`标识在当前数据库中可用的字符集。 因为PostgreSQL不支持在一个数据库中有多个字符集，所以这个视图只能显示一个， 也就是数据库编码。

请注意下列条目在SQL标准中是怎样使用的：

字符指令系统

一个概要的字符集合，例如`UNICODE`，`UCS`，或`LATIN1`。不是作为一个SQL对象公开的，但是在这个视图中可见。

字符编码形式

一些字符指令系统的编码。大多数老旧的字符指令系统只使用一种编码形式，并且因此他们没有单独的名字 （例如`LATIN1`是一种适用于`LATIN1`指令系统的编码形式）。 但是例如Unicode有编码形式`UTF8`，`UTF16`等等。 （不是所有都被PostgreSQL支持）。编码形式不是作为SQL对象公开的，但是在这个视图中可见。

字符集

一个标志字符指令系统的命名的SQL对象，一种字符编码，和一个缺省的排序规则。 一个预定义的字符集通常和编码形式有相同的名称，但是用户可以定义其他名称。 例如，字符集`UTF8`通常识别字符指令系统`UCS`， 编码形式`UTF8`和一些缺省的排序规则。

你可以认为PostgreSQL中的一个"encoding"是一个字符集或一种字符编码形式。 他们将有相同的名称，并且在一个数据库中只能有一个。

Table 34-5. `character_sets` 字段

名字	数据类型	描述
`character_set_catalog`	`sql_identifier`	字符集当前还没有作为模式对象实现，所以这个字段是null。
character_set_schema	sql_identifier	字符集当前还没有作为模式对象实现，所以这个字段是null。
`character_set_name`	`sql_identifier`	字符集的名称，目前是作为数据库编码的显示名称实现的。
character_repertoire	sql_identifier	字符系统指令，如果编码是 UTF8 则显示 UCS ，否则只显示编码名称
`form_of_use`	`sql_identifier`	字符编码形式，和数据库编码相同
default_collate_catalog	sql_identifier	包含缺省排序规则的数据库名（总是当前数据库，如果指定了任意排序规则）
`default_collate_schema`	`sql_identifier`	包含缺省排序规则的模式名
default_collate_name	sql_identifier	缺省排序规则名。缺省排序规则指定为匹配 COLLATE 和 CTYPE 当前数据库设置的规则。如果没有这种规则，那么这个字段和相关的模式和目录字段为null。

34.8. `check_constraint_routine_usage`

视图 `check_constraint_routine_usage` 标识被一个检查约束条件使用的日常活动 （函数和程序）。只有那些属于当前启用角色的日常活动被显示。

Table 34-6. `check_constraint_routine_usage` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含该约束的数据库名字（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含该约束的模式名字
`constraint_name`	`sql_identifier`	约束的名字
`specific_catalog`	`sql_identifier`	包含函数的数据库名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含函数的模式名称
`specific_name`	`sql_identifier`	函数的"专用名"。参阅Section 34.40获取详细信息。

34.9. `check_constraints`

视图 `check_constraints` 包含所有当前用户拥有的检查约束， 可能是定义在表上的，也可能是定义在域上的。（表或者域的所有者就是约束的所有者。）

Table 34-7. `check_constraints` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含此约束的数据库的名字（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含此约束的模式的名字
`constraint_name`	`sql_identifier`	约束的名字
`check_clause`	`character_data`	检查约束的检查表达式

34.10. `collations`

视图 `collations` 包含当前数据库可用的排序规则。

Table 34-8. `collations` 字段

名字	数据类型	描述
`collation_catalog`	`sql_identifier`	包含该排序规则的数据库的名字（总是当前数据库）
`collation_schema`	`sql_identifier`	包含该排序规则的模式的名字
`collation_name`	`sql_identifier`	缺省排序规则的名字
`pad_attribute`	`character_data`	总是 `NO PAD` （PostgreSQL不支持可替代的 `PAD SPACE` 。）

34.11. `collation_character_set_applicability`

视图`collation_character_set_applicability`指定可用的排序规则适用于哪个字符集。 在PostgreSQL中，每个数据库只有一个字符集（参阅Section 34.7里面的说明）， 所以该视图并不提供多少有用的信息。

Table 34-9. `collation_character_set_applicability` 字段

名字	数据类型	描述
`collation_catalog`	`sql_identifier`	包含该排序规则的数据库的名字（总是当前数据库）
`collation_schema`	`sql_identifier`	包含该排序规则的模式的名字
`collation_name`	`sql_identifier`	缺省排序规则的名字
`character_set_catalog`	`sql_identifier`	字符集当前没有作为模式对象实现，所以这个字段是null。
`character_set_schema`	`sql_identifier`	字符集当前没有作为模式对象实现，所以这个字段是null。
`character_set_name`	`sql_identifier`	字符集的名字

34.12. `column_domain_usage`

视图 `column_domain_usage` 标识所有使用了一些域的字段（表或者视图的）， 这些域是在当前数据库中定义的并且是当前用户拥有的。

Table 34-10. `column_domain_usage` 字段

名字	数据类型	描述
`domain_catalog`	`sql_identifier`	包含该域的数据库的名字（总是当前数据库）
`domain_schema`	`sql_identifier`	包含该域的模式的名字
`domain_name`	`sql_identifier`	该域的名字
`table_catalog`	`sql_identifier`	包含该表的数据库的名字（总是当前数据库）
`table_schema`	`sql_identifier`	包含此表的模式名字
`table_name`	`sql_identifier`	表的名字
`column_name`	`sql_identifier`	字段的名字

34.13. `column_options`

视图 `column_options` 包含所有在当前数据库中为外表字段定义的选项。 当前用户登录后只显示这些外表字段（通过成为所有者或拥有某些权限）。

Table 34-11. `column_options` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含该外表的数据库的名字（总是当前数据库）
`table_schema`	`sql_identifier`	包含该外表的模式的名字
`table_name`	`sql_identifier`	外表的名字
`column_name`	`sql_identifier`	字段的名字
`option_name`	`sql_identifier`	选项的名字
`option_value`	`character_data`	选项的值

34.14. `column_privileges`

视图 `column_privileges` 标出所有在当前用户的字段上赋予的权限或者当前用户赋予的字段的权限。 每个字段、授权人和被赋予权利的用户组成一行。

如果权限被赋予整个表，它将在这个视图中显示为每一个字段赋权，但是它只为字段粒度有意义的权限类型： `SELECT` ， `INSERT` ， `UPDATE` ， `REFERENCES` 。

Table 34-12. `column_privileges` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	赋予权限的用户的名字
`grantee`	`sql_identifier`	被赋予权限的用户的名字
`table_catalog`	`sql_identifier`	包含该字段的表所在的数据库名字（总是当前数据库）
`table_schema`	`sql_identifier`	包含该字段的表所在的模式的名字
`table_name`	`sql_identifier`	包含该字段的表的名字
`column_name`	`sql_identifier`	该字段的名字
`privilege_type`	`character_data`	权限类型： `SELECT` ， `INSERT` ， `UPDATE` ，或者 `REFERENCES`
`is_grantable`	`yes_or_no`	如果权限可以赋予， 为 `YES` ， 否则为 `NO`

34.15. `column_udt_usage`

视图 `column_udt_usage` 标出所有使用属于当前用户的数据类型的字段。 请注意，在PostgreSQL里，内置的数据类型的行为和用户定义的类型相似， 因此它们也在这里包括进来了。参阅Section 34.16获取细节。

Table 34-13. `column_udt_usage` 字段

名字	数据类型	描述
`udt_catalog`	`sql_identifier`	这个字段数据类型（如果适用，就是域的基础类型）定义所在的数据库名字（总是当前数据库）。
`udt_schema`	`sql_identifier`	字段数据类型（如果适用，就是域的基础类型）定义所在的模式名字。
`udt_name`	`sql_identifier`	字段数据类型的名称（如果适用，就是域的基础类型）
`table_catalog`	`sql_identifier`	包含该表的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	包含该表的模式的名称
`table_name`	`sql_identifier`	表的名字
`column_name`	`sql_identifier`	字段的名字

34.16. `columns`

视图 `columns` 包含有关数据库中所有表字段（或者视图字段）的信息。 不包括系统字段（比如 `oid` 等）。只有那些当前用户有权访问的字段才会显示出来 （要么是所有者，要么是有些权限）。

Table 34-14. `columns` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含表的数据库的名字（总是当前数据库）
`table_schema`	`sql_identifier`	包含表的模式的名字
`table_name`	`sql_identifier`	表的名字
`column_name`	`sql_identifier`	字段的名字
`ordinal_position`	`cardinal_number`	字段在表中的位置序号（从 1 开始）
`column_default`	`character_data`	字段的缺省表达式
`is_nullable`	`yes_or_no`	如果字段可能为空，则为 `YES` ，如果知道它不能为空，则为 `NO` 。 非空约束是我们得知字段不能为空的一个手段，但是还可能有其它的。
`data_type`	`character_data`	如果它是一个内置类型，那么为字段的数据类型，如果它是某种数组， 则为 `ARRAY` （在这种情况下，参阅视图 `element_types` ）， 否则就是 `USER-DEFINED` （这时， 类型定义在 `udt_name` 和相关的字段上）。 如果字段基于域，这个字段引用底层域类型（而域是在 `domain_name` 和相关字段里定义的）。
`character_maximum_length`	`cardinal_number`	如果 `data_type` 标识一个字符或者位串类型，那么就是声明的最大长度；如果是其它类型或者没有定义最大长度，就是空。
`character_octet_length`	`cardinal_number`	如果 `data_type` 标识一个字符类型，就是以字节记的最大可能长度； 所有其它类型都是空。最大字节长度取决于声明的字节最大长度（见上文）和服务器编码。
		如果 `data_type` 标识一个数值类型，这个字段包含 （声明的或隐含的）这个字段的数据类型的精度。精度表示

`numeric_precision`	`cardinal_number`	有效小数位的长度。 它可以用十进制或者二进制来表示，这一点在 `numeric_precision_radix` 字段里声明。对于其它数据类型，这个字段是空。
`numeric_precision_radix`	`cardinal_number`	如果 `data_type` 标识一个数值类型，这个字段标识字段 `numeric_precision` 和 `numeric_scale` 里的数据是多少进制的。值要么是 2 要么是 10。对于所有其它数据类型，这个字段是空。
`numeric_scale`	`cardinal_number`	如果 `data_type` 标识一个精确的数值类型， 那么这个字段包含（声明的或者隐含的）这个字段上这个类型的数值范围。 数值范围表明小数点右边的有效小数位的数目。它可以用十进制（10为基） 或者二进制（二为基）来表示，正如字段 `numeric_precision_radix` 声明的那样。对于所有其它数据类型，这个字段是空。
`datetime_precision`	`cardinal_number`	如果 `data_type` 标识一个日期，时间，时间戳，或者间隔类型， 该字段（公开地或隐含地）包含该字段类型的小数秒精度，就是说， 小数位数保持到小数点后面。对于所有其它的数据类型，该字段是null。
`interval_type`	`character_data`	如果 `data_type` 标识一个间隔类型，这个字段包含这个字段时间间隔声明， 例如， `YEAR TO MONTH` , `DAY TO SECOND` 等等。 如果没有指定字段限制（也就是，间隔接受所有字段），或对于所有其他数据类型，这个字段是null。
`interval_precision`	`cardinal_number`	用于一个PostgreSQL不可用的特性（参阅 `datetime_precision` 获取间隔类型字段的小数秒精度）
`character_set_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`character_set_schema`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`character_set_name`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`collation_catalog`	`sql_identifier`	包含该字段的排序规则的数据库的名字（总是当前数据库），缺省或者字段的数据类型不可排序时为null。

`collation_schema`	`sql_identifier`	包含该字段的排序规则的模式的名字，缺省或者字段的数据类型不可排序时为null。
`collation_name`	`sql_identifier`	字段的排序规则的名字，缺省或者字段的数据类型不可排序时为null。
`domain_catalog`	`sql_identifier`	如果字段是域类型，就是该域定义所在的数据库的名字（总是当前数据库），否则为null。
`domain_schema`	`sql_identifier`	如果字段是域类型，就是域定义所在的模式的名字，否则为null。
`domain_name`	`sql_identifier`	如果字段是域类型，就是该域的名字，否则为null。
`udt_catalog`	`sql_identifier`	这个字段数据类型（如果适用，就是底层域类型）定义所在的数据库的名字（总是当前数据库）。
`udt_schema`	`sql_identifier`	这个字段数据类型（如果适用，就是底层域类型）定义所在的模式名字。
`udt_name`	`sql_identifier`	这个字段数据类型（如果适用，就是底层域类型）的名字。
`scope_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`scope_schema`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`scope_name`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`maximum_cardinality`	`cardinal_number`	总是空，因为在PostgreSQL里数组总是有无限的最大维数
`dtd_identifier`	`sql_identifier`	一个该字段的数据类型描述符的标识符，在属于这个表中的所有的数据类型描述符中唯一。 这个字段主要用于和其它这样的标识符实例连接。 （这个标识符的确切格式没有定义并且不保证在将来的版本中保持一样。）
`is_self_referencing`	`yes_or_no`	用于PostgreSQL里一个不可用的特性
`is_identity`	`yes_or_no`	用于PostgreSQL里一个不可用的特性
`identity_generation`	`character_data`	用于PostgreSQL里一个不可用的特性
`identity_start`	`character_data`	用于PostgreSQL里一个不可用的特性

identity_increment	character_data	用于PostgreSQL里一个不可用的特性
identity_maximum	character_data	用于PostgreSQL里一个不可用的特性
identity_minimum	character_data	用于PostgreSQL里一个不可用的特性
identity_cycle	yes_or_no	用于PostgreSQL里一个不可用的特性
is_generated	character_data	用于PostgreSQL里一个不可用的特性
generation_expression	character_data	用于PostgreSQL里一个不可用的特性
is_updatable	yes_or_no	如果字段为可更新则为 YES ，否则为 NO （基表中的字段总是可以更新的，而试图中的字段则不一定）

因为数据类型在SQL里可以用多种方法定义，并且PostgreSQL包含额外的定义数据类型的方法，因此他们在信息模式里的表现形式可能不太一样。字段 `data_type` 会被用于标识该字段底层的内置数据类型。在PostgreSQL里，这意味着类型将定义在系统表模式 `pg_catalog` 里。如果应用可以很好地处理那些重要的内置类型（比如，对数值类型格式化成不同的东西，或者使用在精度字段里的数据），那么这个字段是有用的。字段 `udt_name` ，`udt_schema` ，和 `udt_catalog` 总是标识该字段的底层数据类型，即使字段是基于域的也一样。（因为PostgreSQL 把内置类型看作和用户定义类型一样，所以，内置类型也在这里出现。这是对SQL标准的一个扩展。）如果一个应用想根据数据类型的不同而区别处理数据，那么应该使用这些字段，因为在这种情况下它不会在意这个字段是否真正基于域的。如果这个字段基于一个域，那么该域的标识保存在字段 `domain_name` ，`domain_schema` ，和 `domain_catalog` 里。如果你想把字段和他们相关的数据类型凑成对儿，并且把域当作不同的类型处理，你可以这么写 `coalesce(domain_name,udt_name)` 等等。

34.17. `constraint_column_usage`

视图 `constraint_column_usage` 标识在当前数据库中使用了某种约束的所有字段。 只有那些属于当前用户的表中的字段才会被显示出来。对于检查约束，这个视图标识用在检查表达式里的字段。 对于外键约束，这个视图标识外键引用的字段。对于唯一或主键约束，这个视图标识被约束的字段。

Table 34-15. `constraint_column_usage` Columns

名字	数据类型	描述
`table_catalog`	`sql_identifier`	被某个（些）约束使用的字段所在的表所在的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	被某个（些）约束使用的字段所在的表所在的模式名
`table_name`	`sql_identifier`	被某个（些）约束使用的字段所在的表名
`column_name`	`sql_identifier`	被某个（些）约束使用的字段名
`constraint_catalog`	`sql_identifier`	包含该约束的数据库名（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含该约束的模式名
`constraint_name`	`sql_identifier`	约束名

34.18. `constraint_table_usage`

视图 `constraint_table_usage` 标识当前数据库中被某些约束使用并且被当前用户拥有的所有表。（它和视图 `table_constraints` 不同，这个视图标识所有约束以及他们定义所在的表。）对于一个外键约束，这个事务标识外键引用的表。对于唯一或者主键约束，这个视图只是简单标识这个约束所属的表。 检查约束和非空约束没有包含在这个视图中。

Table 34-16. `constraint_table_usage` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含被某些约束使用的表的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	包含被某些约束使用的表的模式的字
`table_name`	`sql_identifier`	被某些约束使用的表名
`constraint_catalog`	`sql_identifier`	包含该约束的数据库名（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含约束的模式名
`constraint_name`	`sql_identifier`	约束名

34.19. `data_type_privileges`

视图 `data_type_privileges` 标识当前用户可访问的所有数据类型描述符， 只要他是被描述对象的所有者或者有某些权限。一个数据类型描述符是在一种数据类型用在表字段， 域，或者函数（参数或者返回类型）定义的时候生成的，它包含一些有关这个数据类型在该实例中如何使用的信息 （比如，声明的最大长度——如果有的话）。每种数据类型描述符都赋予了一个任意的标识符， 这个标识符在赋予某个对象 （表，域，函数）的所有数据类型描述符标识符中唯一。 这个视图可能对应用没什么用，但是用于在信息模式中定义一些其它的视图。

Table 34-17. `data_type_privileges` 字段

名字	数据类型	描述
`object_catalog`	`sql_identifier`	包含被描述的对象的数据库名（总是当前数据库）
`object_schema`	`sql_identifier`	包含被描述对象的模式名
`object_name`	`sql_identifier`	被描述对象的名字
`object_type`	`character_data`	被描述对象的类型： `TABLE` （数据类型描述符属于该表的一个字段）， `DOMAIN` 数据类型描述符属于该域）， `ROUTINE` （数据类型描述符属于该函数的一个参数或者返回的数据类型）之一。
`dtd_identifier`	`sql_identifier`	这个数据类型描述符的标识符，在用于同一个对象上的所有数据类型描述符中是唯一的。

34.20. `domain_constraints`

视图 `domain_constraints` 包含属于所有在当前数据库中定义的域的约束。 只有当前用户有权访问的域才显示（通过成为其所有者或有某些权限）。

Table 34-18. `domain_constraints` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含此约束的数据库名（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含此约束的模式名
`constraint_name`	`sql_identifier`	约束名
`domain_catalog`	`sql_identifier`	包含该域的数据库名（总是当前数据库）
`domain_schema`	`sql_identifier`	包含该域的模式名
`domain_name`	`sql_identifier`	域名
`is_deferrable`	`yes_or_no`	如果约束可以推迟，则为 `YES` ，如果不行，则为 `NO`
`initially_deferred`	`yes_or_no`	如果约束可以推迟，且为初始推迟，则为 `YES` ，否则为 `NO` 。

34.21. `domain_udt_usage`

视图 `domain_udt_usage` 标识所有使用当前用户拥有的数据类型的域。 请注意，在PostgreSQL里，内置数据类型的行为和用户定义的类型一样， 因此它们也在这里包含。

Table 34-19. `domain_udt_usage` 字段

名字	数据类型	描述
`udt_catalog`	`sql_identifier`	该域数据类型定义所在的数据库名（总是当前数据库）
`udt_schema`	`sql_identifier`	这个域数据类型定义所在的模式名
`udt_name`	`sql_identifier`	域数据类型名
`domain_catalog`	`sql_identifier`	包含该域的数据库名（总是当前数据库）
`domain_schema`	`sql_identifier`	包含该域的模式名
`domain_name`	`sql_identifier`	域名

34.22. `domains`

视图 `domains` 包含定义在当前数据库中的所有域。 只有当前用户有权访问的域才显示（通过成为其所有者或有某些权限）。

Table 34-20. `domains` 字段

名字	数据类型	描述
`domain_catalog`	`sql_identifier`	包含这个域的数据库名（总是当前数据库）
`domain_schema`	`sql_identifier`	包含这个域的模式名
`domain_name`	`sql_identifier`	域的名字
`data_type`	`character_data`	如果这是一个内置类型，就是域的数据类型，如果他是某种数组（这时，参阅视图 `element_types` ），就是 `ARRAY` 。 否则是 `USER-DEFINED` （这个时候，类型在 `udt_name` 和相关的字段里面标识。）
`character_maximum_length`	`cardinal_number`	如果域是一个字符或者位串类型，这是定义的最大长度，其它数据类型或者没有声明最大长度，则为空。
`character_octet_length`	`cardinal_number`	如果域有一个字符类型，这是最大可能的字节长度；其它所有的数据类型则为null。 最大字节长度取决于所声明的字符最大长度（见上文）和服务器的编码。
`character_set_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`character_set_schema`	`sql_identifier`	
`character_set_name`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`collation_catalog`	`sql_identifier`	包含域的排序规则的数据库的名字（总是当前数据库），如果缺省或域的数据类型不可排序则为null。
`collation_schema`	`sql_identifier`	包含域的排序规则的模式的名字，如果缺省或域的数据类型不可排序则为null。
`collation_name`	`sql_identifier`	域的排序规则的名字，如果缺省或者域的数据类型不可排序则为null。
		如果这个域有一个数值类型，那么这个字段包含（声明的或隐含的）用于这个字段的类型精度。 精度表示有效

`numeric_precision`	`cardinal_number`	数据位的个数。可以用十进制表示，也可以用二进制表示，就像在 `numeric_precision_radix` 字段里声明的那样。对于所有其它类型，这个字段是空。
`numeric_precision_radix`	`cardinal_number`	如果这个域有一个数值类型，那么这个字段标识 `numeric_precision` 和 `numeric_scale` 字段的数值的基数是多少。数值要么是 2 要么是 10。 对于所有其它数据类型，这个字段是空。
`numeric_scale`	`cardinal_number`	如果这个域有一个准确的数值类型，那么这个字段包含（声明或者隐含的）本字段该类型的数值范围。 数值范围标志着小数点右边的有效位数。它可以是用十进制表示，也可以用二进制表示， 就像在 `numeric_precision_radix` 字段里声明的那样。对于所有其它数据类型，这个字段是空。
`datetime_precision`	`cardinal_number`	如果 `data_type` 标识一个日期，时间，时间戳，或者间距类型， 该字段（明确的或隐含地）包含部分的域类型秒精度，就是说，小数位数保持到小数点后面。 对于所有其它的数据类型， 该字段是null。
`interval_type`	`character_data`	如果 `data_type` 标识间隔类型，这个字段包含这个领域包括的字段的时间间隔的声明， 例如 `YEAR TO MONTH`， `DAY TO SECOND` 等等。 如果没有声明字段限制（也就是， 间隔接受所有字段），或是对于所有其他类型，这个字段为null。
`interval_precision`	`cardinal_number`	适用于PostgreSQL里不可用的一个特性 （参阅 `datetime_precision` 获取间隔类型域的小数秒精度）
`domain_default`	`character_data`	这个域的缺省表达式
`udt_catalog`	`sql_identifier`	域数据类型定义所在的数据库名（总是当前数据库）
`udt_schema`	`sql_identifier`	域数据类型定义所在的模式名
`udt_name`	`sql_identifier`	域数据类型名
`scope_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`scope_schema`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`scope_name`	`sql_identifier`	用于PostgreSQL里一个不可用的特性

`maximum_cardinality`	`cardinal_number`	总是为空，因为PostgreSQL的数组总是有无限的维数。
`dtd_identifier`	`sql_identifier`	一个该域的数据类型描述符的标识符，在属于该域的所有数据类型描述符中是唯一的（这个是琐事，因为一个域只包含一种数据类型）。这个主要用于和其它这样的标识符实例连接。（这个标识符的具体格式没有定义，并且不保证在将来版本中保持一致。）

34.23. `element_types`

视图 `element_types` 包含数组元素的数据类型的描述符。在一个表字段，复合类型属性，域，函数参数，或者函数返回值定义为一个数组类型的时候，对应的信息模式视图在字段 `data_type` 里只包含 `ARRAY` 。要获取数组元素类型的信息，你可以将对应的视图和这个视图连接。 比如，要用数据类型和数组元素类型显示表的字段，（如果可能），你可以用：

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.collection_t
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

这个视图只包含当前用户具有权限访问的对象，通过成为其所有者或者有些权限。

Table 34-21. `element_types` 字段

名字	数据类型	描述
`object_catalog`	`sql_identifier`	使用了被描述的数组的对象所在的数据库名（总是当前数据库）
`object_schema`	`sql_identifier`	使用了被描述的数组的对象所在的模式名
`object_name`	`sql_identifier`	使用了被描述的数组的对象的名字
`object_type`	`character_data`	使用了被描述的数组的对象的类型： `TABLE` （数组被该表的一个字段使用）， `USER-DEFINED TYPE` （数组被这个复合类型的属性使用）， `DOMAIN` （数组被这个域使用）， `ROUTINE` （数组被该函数的一个参数或者返回数据类型使用）之一。
`collection_type_identifier`	`sql_identifier`	被描述的数组的数据类型描述符的标识符。使用它与其他信息模式视图的 `dtd_identifier` 字段连接。
`data_type`	`character_data`	如果这是一个内置的类型，数据元素的数据类型，否则就是 `USER-DEFINED` （这种情况下，类型在 `udt_name` 和相关的字段中标出）。
`character_maximum_length`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类

		型。
`character_octet_length`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型。
`character_set_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`character_set_schema`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`character_set_name`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`collation_catalog`	`sql_identifier`	包含元素类型的排序规则的数据库的名字（总是当前数据库），如果缺省或元素的数据类型不可排序则为null。
`collation_schema`	`sql_identifier`	包含元素类型的排序规则的模式的名字，如果缺省或元素的数据类型不可排序则为null。
`collation_name`	`sql_identifier`	元素类型的排序规则的名字，如果缺省或元素的数据类型不可排序则为null。
`numeric_precision`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`numeric_precision_radix`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`numeric_scale`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`datetime_precision`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`interval_type`	`character_data`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`interval_precision`	`cardinal_number`	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
`domain_default`	`character_data`	未实现
`udt_catalog`	`sql_identifier`	元素的数据类型定义所在的数据库名（总是当前数据库）
`udt_schema`	`sql_identifier`	元素的数据类型定义所在的模式名
`udt_name`	`sql_identifier`	元素的数据类型名
`scope_catalog`	`sql_identifier`	用于PostgreSQL里一个不可用的特性
`scope_schema`	`sql_identifier`	用于PostgreSQL里一个不可用的特

scope_schema	sql_identifier	性
scope_name	sql_identifier	用于PostgreSQL里一个不可用的特性
maximum_cardinality	cardinal_number	总是空，因为这个信息并不适用于PostgreSQL里的数组元素数据类型
dtd_identifier	sql_identifier	元素的数据类型描述符的标识符。目前没什么用处。

34.24. `enabled_roles`

视图 `enabled_roles` 标识当前 "已授权角色"。 授权的角色递归地被定义为当前用户和所有通过自动继承授权的角色。换句话说， 当前用户直接地或间接地，自动地继承成员权限。

对于权限的检查，"可用角色"的设置是比较实用的， 可以比启用角色组更广泛。 所以通常，使用视图 `applicable_roles` 会更好一些。

Table 34-22. `enabled_roles` 字段

名字	数据类型	描述
`role_name`	`sql_identifier`	角色名

34.25. `foreign_data_wrapper_options`

视图 `foreign_data_wrapper_options` 包含了当前数据库中为外部数据封装器定义的所有选项。仅有那些当前用户有权访问的外部数据封装器才显示（通过成为所有者或者拥有一些权限来）。

Table 34-23. `foreign_data_wrapper_options` 字段

名字	数据类型	描述
`foreign_data_wrapper_catalog`	`sql_identifier`	定义外部数据封装器的数据库名称 (总是在当前数据库)
`foreign_data_wrapper_name`	`sql_identifier`	外部数据封装器的名称
`option_name`	`sql_identifier`	选项名称
`option_value`	`character_data`	选项值

34.26. `foreign_data_wrappers`

视图 `foreign_data_wrappers` 包含所有定义在当前数据库中的外部数据封装器。 仅有那些当前用户有权访问外部数据封装器才显示（通过成为所有者或者拥有一些权限）。

Table 34-24. `foreign_data_wrappers` 字段

名字	数据类型	描述
`foreign_data_wrapper_catalog`	`sql_identifier`	含有外部数据封装器的数据库名称（总是当前数据库）
`foreign_data_wrapper_name`	`sql_identifier`	外部数据封装器名称
`authorization_identifier`	`sql_identifier`	外部服务器所有者名称
`library_name`	`character_data`	执行外部数据封装器的库文件名
`foreign_data_wrapper_language`	`character_data`	执行外部数据封装器所使用的语言

34.27. `foreign_server_options`

视图 `foreign_server_options` 包含了当前数据库中为外部服务器定义的所有选项。 仅有那些当前用户有权访问的外部服务器才显示（通过成为所有者或者拥有一些权限）。

Table 34-25. `foreign_server_options` 字段

名字	数据类型	描述
`foreign_server_catalog`	`sql_identifier`	定义外部服务器的数据库名称(总是当前数据库)
`foreign_server_name`	`sql_identifier`	外部服务器名称
`option_name`	`sql_identifier`	选项名称
`option_value`	`character_data`	选项值

34.28. `foreign_servers`

视图 `foreign_servers` 包含了当前数据库中定义的所有外部服务器。 仅有那些当前用户有权访问的外部服务器才显示（通过成为所有者或者拥有一些权限）。

Table 34-26. `foreign_servers` 字段

名字	数据类型	描述
`foreign_server_catalog`	`sql_identifier`	定义外部服务器的数据库名称(总是当前数据库)
`foreign_server_name`	`sql_identifier`	外部服务器名称
`foreign_data_wrapper_catalog`	`sql_identifier`	含有被外部服务器所使用的外部数据封装器的数据库的名称（总是当前数据库）
`foreign_data_wrapper_name`	`sql_identifier`	被外部服务器所使用的外部数据封装器名称
`foreign_server_type`	`character_data`	外部服务器类型信息，如果在创建时指定
`foreign_server_version`	`character_data`	外部服务器版本信息，如果在创建时指定
`authorization_identifier`	`sql_identifier`	外部服务器所有者名称

34.29. `foreign_table_options`

视图 `foreign_table_options` 包含所有在当前数据库中为外表定义的选项。 只有那些当前用户有权访问的外表才显示（通过成为所有者或有一些权限）。

Table 34-27. `foreign_table_options` 字段

名字	数据类型	描述
`foreign_table_catalog`	`sql_identifier`	包含该外表的数据库的名字（总是当前数据库）
`foreign_table_schema`	`sql_identifier`	包含该外表的模式的名字
`foreign_table_name`	`sql_identifier`	外表的名字
`foreign_server_catalog`	`sql_identifier`	定义外部服务器的数据库的名字（总是当前数据库）
`foreign_server_name`	`sql_identifier`	外部服务器的名字
`option_name`	`sql_identifier`	选项名
`option_value`	`character_data`	选项值

34.30. `foreign_tables`

视图 `foreign_tables` 包含所有在当前数据库中定义的外表。 只有那些当前用户有权访问的外表才显示（通过成为所有者或有一些权限）。

Table 34-28. `foreign_tables` 字段

名字	数据类型	描述
`foreign_table_catalog`	`sql_identifier`	定义外表的数据库的名字（总是当前数据库）
`foreign_table_schema`	`sql_identifier`	包含外表的模式的名字
`foreign_table_name`	`sql_identifier`	外表的名字
`foreign_server_catalog`	`sql_identifier`	定义外部服务器的数据库的名字（总是当前数据库）
`foreign_server_name`	`sql_identifier`	外部服务器的名字

34.31. `key_column_usage`

视图 `key_column_usage` 标出当前数据库中所有被某些唯一约束、 主键约束或者外键约束限制的字段。在这个视图里没有包含检查约束。 只有当前用户可以访问的那些字段才显示，通过成为所有者或有一些权限。

Table 34-29. `key_column_usage` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含这个约束的数据库的名称（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含这个约束的模式的名称
`constraint_name`	`sql_identifier`	约束的名称
`table_catalog`	`sql_identifier`	包含被这个约束限制着某个字段的表所在的数据库的名称（总是当前数据库）
`table_schema`	`sql_identifier`	包含被这个约束限制着某个字段的表所在的模式的名称
`table_name`	`sql_identifier`	包含被这个约束限制着某个字段的表的名称
`column_name`	`sql_identifier`	被这个约束限制的字段的名称
`ordinal_position`	`cardinal_number`	字段在约束键字里的位置序号（从 1 开始）
`position_in_unique_constraint`	`cardinal_number`	对于一个外键约束，唯一约束中引用行的顺序位置(从1开始)；否则为 null

34.32. `parameters`

视图 `parameters` 包含有关当前数据库里所有函数的参数的信息。 只有当前用户有访问权限的函数才会在这里显示出来（用户要么是所有者，要么有些权限）。

Table 34-30. `parameters` Columns

名字	数据类型	描述
`specific_catalog`	`sql_identifier`	包含此函数的数据库的名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含此函数的模式的名字
`specific_name`	`sql_identifier`	函数的"specific name"（具体名称）。 参阅Section 34.40获取更多信息。
`ordinal_position`	`cardinal_number`	参数在函数的参数列表里的位置序号（从 1 开始）
`parameter_mode`	`character_data`	`IN` 用于输入的参数， `OUT` 用于输出的参数， 和 `INOUT` 用于输入输出的参数。
`is_result`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
`as_locator`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
`parameter_name`	`sql_identifier`	参数名称，如果参数没有名称则为空
`data_type`	`character_data`	如果是内置类型，那么是参数的数据类型，如果它是某种数组就是 `ARRAY` （这种情况下，参阅视图 `element_types` ），否则就是 `USER-DEFINED` （这种情况下，该类型在 `udt_name` 和相关的字段中标出）。
`character_maximum_length`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`character_octet_length`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`character_set_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`character_set_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
		应用于一个PostgreSQL里没有的特

		性
`collation_catalog`	`sql_identifier`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`collation_schema`	`sql_identifier`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`collation_name`	`sql_identifier`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`numeric_precision`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`numeric_precision_radix`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`numeric_scale`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`datetime_precision`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`interval_type`	`character_data`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`interval_precision`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`udt_catalog`	`sql_identifier`	该参数数据类型定义所在的数据库名称（总是当前数据库）
`udt_schema`	`sql_identifier`	该参数数据类型定义所在的模式名称
`udt_name`	`sql_identifier`	参数的数据类型名称
`scope_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`scope_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`scope_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`maximum_cardinality`	`cardinal_number`	总是空值，因为这个信息不适用于PostgreSQL里的参数数据类型
`dtd_identifier`	`sql_identifier`	参数的数据类型描述符的标识符，在属于该函数的所有数据类型描述符中唯一。 这个字段主要用于可以和这样的其它标识符实例进行连接。 （这个标识符的具体格式没有在标准中定义，并且并不保证在将来的版本中保持一致。）

34.33. `referential_constraints`

视图 `referential_constraints` 包含当前数据库里的所有参考（外键）约束。 只有当前用户有权访问的引用表里的约束才显示（通过成为所有者或有某些权限，而不是 `SELECT` ）。

Table 34-31. `referential_constraints` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含这个约束的数据库名字（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含这个约束的模式名字
`constraint_name`	`sql_identifier`	这个约束的名字
`unique_constraint_catalog`	`sql_identifier`	包含该外键约束引用的唯一或者主键约束的数据库名称（总是当前数据库）
`unique_constraint_schema`	`sql_identifier`	该外键约束引用的唯一或者主键约束的模式的名称
`unique_constraint_name`	`sql_identifier`	该外键约束引用的唯一或者主键约束的名称
`match_option`	`character_data`	该外键约束的匹配选项： `FULL` , `PARTIAL` ,或者 `NONE` 。
`update_rule`	`character_data`	这个外键约束的更新规则： `CASCADE` , `SET NULL` , `SET DEFAULT` , `RESTRICT` 或者 `NO ACTION` 。
`delete_rule`	`character_data`	这个外键约束的删除规则 `CASCADE` , `SET NULL` , `SET DEFAULT` , `RESTRICT` 或者 `NO ACTION` 。

34.34. role_column_grants

视图 role_column_grants 标识那些在字段上赋予或被赋予为当前角色的所有权限。 更多信息可以在 column_privileges 中找到。这个视图和 column_privileges 唯一有效的不同就是这个视图忽略了某些字段，这些字段是通过授予 PUBLIC 使得当前用户可以访问的字段。

Table 34-32. role_column_grants 字段

名字	数据类型	描述
grantor	sql_identifier	被赋予这个权限的用户名称
grantee	sql_identifier	被赋予这个权限的角色名称
table_catalog	sql_identifier	包含此字段的表所在的数据库的名字（总是当前数据库）
table_schema	sql_identifier	包含该字段的表所在模式的名称
table_name	sql_identifier	包含该字段的表名称
column_name	sql_identifier	该字段的名称
privilege_type	character_data	权限的类型：SELECT，INSERT，UPDATE 或者 REFERENCES
is_grantable	yes_or_no	如果权限是可以赋予的，则为 YES ，否则，为 NO

34.35. `role_routine_grants`

视图 `role_routine_grants` 标出所有在函数上赋予或被赋予当前角色的权限。 更多的信息可以在 `routine_privileges` 里找到。在该视图与 `routine_privileges` 之间实际仅有的差异是该视图忽略那些通过赋权给 `PUBLIC` 使当前用户可以访问的函数。

Table 34-33. `role_routine_grants` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	被赋予该权限的角色名称
`grantee`	`sql_identifier`	被赋予此权限的角色的名称
`specific_catalog`	`sql_identifier`	包含此函数的数据库名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含此函数的模式名称
`specific_name`	`sql_identifier`	函数的"specific name"（具体的名称）。 参阅Section 34.40获取更多信息。
`routine_catalog`	`sql_identifier`	包含此函数的数据库名称（总是当前数据库）
`routine_schema`	`sql_identifier`	包含此函数的模式名称
`routine_name`	`sql_identifier`	函数的名称（可能重复，因为有重载）
`privilege_type`	`character_data`	总是 `EXECUTE` （函数的唯一的权限类型）
`is_grantable`	`yes_or_no`	如果权限可以赋予，那么是 `YES` ， 否则为 `NO`

34.36. `role_table_grants`

视图 `role_table_grants` 标识在表或者视图上赋予或被赋予当前角色的全部权限。 更多信息可以在 `table_privileges` 找到。在该视图与 `table_privileges` 之间实际仅有的差异是该视图忽略那些通过赋权给 `PUBLIC` 使当前用户可以访问的表。

Table 34-34. `role_table_grants` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	赋予权限的角色名
`grantee`	`sql_identifier`	被赋予权限的角色名
`table_catalog`	`sql_identifier`	包含此表的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	包含此表的模式名
`table_name`	`sql_identifier`	表名
`privilege_type`	`character_data`	权限类型： `SELECT` ， `INSERT` ， `UPDATE` ， `DELETE` ， `TRUNCATE` ， `REFERENCES` ，或 `TRIGGER`
`is_grantable`	`yes_or_no`	如果权限可以赋予则为 `YES` ， 否则为 `NO`
`with_hierarchy`	`yes_or_no`	在SQL标准里， `WITH HIERARCHY OPTION` 是一个分开的（子）权限，允许在表继承层次结构上进行创建操作。 在PostgreSQL中，这包含在了 `SELECT` 权限中， 所以如果这个权限为 `SELECT` 则这个字段显示 `YES` ， 否则为 `NO` 。

34.37. `role_udt_grants`

视图 `role_udt_grants` 用于标出赋予或被赋予当前角色的用户定义类型上的 `USAGE` 权限。更多的信息可以在 `udt_privileges` 里找到。 在该视图与 `udt_privileges` 之间实际仅有的差异是该视图忽略那些通过赋权给 `PUBLIC` 使当前用户可以访问的对象。 因为数据类型在PostgreSQL中并没有真实的权力，但是只有一个隐式的赋权给 `PUBLIC`， 所以这个视图是空的。

Table 34-35. `role_udt_grants` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	赋予该权限的角色的名字
`grantee`	`sql_identifier`	被赋予该权限的角色的名字
`udt_catalog`	`sql_identifier`	包含该类型的数据库的名字（总是当前数据库）
`udt_schema`	`sql_identifier`	包含该类型的模式的名字
`udt_name`	`sql_identifier`	类型名
`privilege_type`	`character_data`	总是 `TYPE USAGE`
`is_grantable`	`yes_or_no`	如果权限是可赋予的，那么就是 `YES`， 否则为 `NO`

34.38. `role_usage_grants`

视图 `role_usage_grants` 用于标出当前角色赋予或被赋予的各种对象的 `USAGE` 权限。更多的信息可以在 `usage_privileges` 里找到。 在该视图与 `usage_privileges` 之间实际仅有的差异是该视图忽略那些通过赋权给 `PUBLIC` 使当前用户可以访问的对象。

Table 34-36. `role_usage_grants` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	赋予该权限的角色名称
`grantee`	`sql_identifier`	被赋予该权限的角色的名称
`object_catalog`	`sql_identifier`	包含该对象的数据库的名字（总是当前数据库）
`object_schema`	`sql_identifier`	如果适用，是包含该对象的模式的名字，否则是一个空字符串
`object_name`	`sql_identifier`	对象的名字
`object_type`	`character_data`	`COLLATION` 或 `DOMAIN` 或 `FOREIGN DATA WRAPPER` 或 `FOREIGN SERVER` 或 `SEQUENCE`
`privilege_type`	`character_data`	Always `USAGE`
`is_grantable`	`yes_or_no`	如果权限可以赋予，则为 `YES` ，否则为 `NO`

34.39. `routine_privileges`

视图 `routine_privileges` 标识在函数上所有赋予当前用户或者由当前用户赋予的权限。 每个函数，授权人，和权限接受人的组合都有一行。

Table 34-37. `routine_privileges` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	赋予权限的角色名
`grantee`	`sql_identifier`	被授予权限的角色名
`specific_catalog`	`sql_identifier`	包含该函数的数据库名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含该函数的模式名
`specific_name`	`sql_identifier`	函数的"specific name"（具体的名字）。 参阅Section 34.40获取更多信息。
`routine_catalog`	`sql_identifier`	包含该函数的数据库名称（总是当前数据库）
`routine_schema`	`sql_identifier`	包含该函数的模式名称
`routine_name`	`sql_identifier`	函数名（可能会因重载而重复）
`privilege_type`	`character_data`	总是 `EXECUTE` (用于函数的唯一的权限类型)
`is_grantable`	`yes_or_no`	如果权限是可赋予的，则为 `YES` ，否则为 `NO`

34.40. `routines`

视图 `routines` 包含当前数据库中的所有函数。 只有当前用户有访问权限（可能是所有者或者有特定权限）的函数才显示出来。

Table 34-38. `routines` 字段

名字	数据类型	描述
`specific_catalog`	`sql_identifier`	包含该函数的数据库名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含该函数的模式名称
`specific_name`	`sql_identifier`	函数的"specific name"（具体名字）。这是一个在模式里唯一标识该函数的名字， 即使函数的真是名字是重载的也如此。具体名字的格式没有定义， 我们应该只是用它和其它具体过程名的实例进行比较。
`routine_catalog`	`sql_identifier`	包含该函数的数据库名称（总是当前数据库）
`routine_schema`	`sql_identifier`	包含该函数的模式名称
`routine_name`	`sql_identifier`	函数的名称（在重载的时候可能重复）
`routine_type`	`character_data`	总是 `FUNCTION` （未来可能会有其它过程的类型。）
`module_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`module_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`module_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`udt_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`udt_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`udt_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
		如果这是一个内置类型，

data_type	character_data	则为函数的返回数据类型，或者如果是某种数组，则为 ARRAY （这个时候，参阅视图 element_types ），否则就是 USER-DEFINED （这种情况下，类型在 type_udt_name 和相关字段中标识）。
character_maximum_length	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
character_octet_length	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
character_set_catalog	sql_identifier	应用于一个PostgreSQL里没有的特性
character_set_schema	sql_identifier	应用于一个PostgreSQL里没有的特性
character_set_name	sql_identifier	应用于一个PostgreSQL里没有的特性
collation_catalog	sql_identifier	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
collation_schema	sql_identifier	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
collation_name	sql_identifier	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
numeric_precision	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
numeric_precision_radix	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
numeric_scale	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
datetime_precision	cardinal_number	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
interval_type	character_data	总是为空，因为这个信息并不应用于PostgreSQL里

		的返回数据类型
`interval_precision`	`cardinal_number`	总是为空，因为这个信息并不应用于PostgreSQL里的返回数据类型
`type_udt_catalog`	`sql_identifier`	函数的返回数据类型定义所在的数据库名称（总是当前数据库）
`type_udt_schema`	`sql_identifier`	函数的返回数据类型定义所在的模式名称
`type_udt_name`	`sql_identifier`	该函数的返回数据类型的名称
`scope_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`scope_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`scope_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`maximum_cardinality`	`cardinal_number`	总是为空，因为数组在PostgreSQL中总是有无限的最大维数
`dtd_identifier`	`sql_identifier`	一个这个函数返回的数据类型的数据类型描述符的标识符， 在所有属于这个函数的数据类型描述符中唯一。这个描述符主要用于和其它这样的标识符实例进行连接。（标识符具体的格式没有定义，并且不保证在将来的版本中保持相同。）
`routine_body`	`character_data`	如果函数是 SQL 函数，那么 `SQL` ，否则是 `EXTERNAL` 。
`routine_definition`	`character_data`	函数的源代码文本（如果当前用户不是函数所有者，则为空）。（根据 SQL 标准， 这个字段只又在 `routine_body` 是 `SQL` 的时候才使用， 但是在PostgreSQL里，这个字段将包含创建函数的时候所声明的任何源文本。）
		如果这个函数是一个 C 函数，那么是函数的外部名字（链接符号）；否则为

		空。（这个字段的数值和 `routine_definition` 里显示的数值相同。）
external_language	character_data	书写这个函数使用的语言
`parameter_style`	`character_data`	总是 `GENERAL` （SQL标准定义了其它参数类型， 那些类型不适用于PostgreSQL。）
is_deterministic	yes_or_no	如果这个函数声明为不变的(immutable)（在 SQL 标准里叫确定的(deterministic)）， 那么是 YES ， 否则是 NO 。（在PostgreSQL里你无法通过信息模式查询其它可用的易失性级别。）
`sql_data_access`	`character_data`	总是 `MODIFIES` ， 意思是这个函数可能修改 SQL 数据。 这个信息对PostgreSQL没啥作用。
is_null_call	yes_or_no	如果当函数任意输入参数为空时函数自动返回空， 则为 YES ， 否则为 NO 。
`sql_path`	`character_data`	应用于一个PostgreSQL里没有的特性
schema_level_routine	yes_or_no	总是 YES （相反的是一个用户定义类型的方法， 这是一个PostgreSQL里没有的特性。）
`max_dynamic_result_sets`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
is_user_defined_cast	yes_or_no	应用于一个PostgreSQL里没有的特性
`is_implicitly_invocable`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
security_type	character_data	如果这个函数以当前用户的权限运行， 则为 INVOKER ， 如果函数以定义它的用户的权限运行， 则为 DEFINER 。
`to_sql_specific_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
to_sql_specific_schema	sql_identifier	应用于一个PostgreSQL里没有的特性

		没有的特性
`to_sql_specific_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`as_locator`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
`created`	`time_stamp`	应用于一个PostgreSQL里没有的特性
`last_altered`	`time_stamp`	应用于一个PostgreSQL里没有的特性
`new_savepoint_level`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
`is_udt_dependent`	`yes_or_no`	当前总是 `NO` 。 `YES` 适用于 PostgreSQL里没有的一个特性。
`result_cast_from_data_type`	`character_data`	应用于一个PostgreSQL里没有的特性
`result_cast_as_locator`	`yes_or_no`	应用于一个PostgreSQL里没有的特性
`result_cast_char_max_length`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
`result_cast_char_octet_length`	`character_data`	应用于一个PostgreSQL里没有的特性
`result_cast_char_set_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_char_set_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_char_set_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_collation_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_collation_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_collation_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_numeric_precision`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
`result_cast_numeric_precision_radix`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
`result_cast_numeric_scale`	`cardinal_number`	应用于一个PostgreSQL里没有的特性

`result_cast_datetime_precision`	`character_data`	应用于一个PostgreSQL里没有的特性
`result_cast_interval_type`	`character_data`	应用于一个PostgreSQL里没有的特性
`result_cast_interval_precision`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
`result_cast_type_udt_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_type_udt_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_type_udt_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_scope_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_scope_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_scope_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`result_cast_maximum_cardinality`	`cardinal_number`	应用于一个PostgreSQL里没有的特性
`result_cast_dtd_identifier`	`sql_identifier`	应用于一个PostgreSQL里没有的特性

34.41. `schemata`

视图 `schemata` 包含当前数据库里由当前用户拥有的所有模式。

Table 34-39. `schemata` 字段

名字	数据类型	描述
`catalog_name`	`sql_identifier`	此模式所在的数据库名称（总是当前数据库）
`schema_name`	`sql_identifier`	模式的名字
`schema_owner`	`sql_identifier`	模式的所有者名称
`default_character_set_catalog`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`default_character_set_schema`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`default_character_set_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`sql_path`	`character_data`	应用于一个PostgreSQL里没有的特性

34.42. `sequences`

视图 `sequences` 包含了所有定义在当前数据库中的序列。 只有那些当前用户可以访问的序列才显示（通过成为所有者或拥有一些权限）。

Table 34-40. `sequences` 字段

名字	数据类型	描述
`sequence_catalog`	`sql_identifier`	包含该序列的数据库名称（总是当前数据库）
`sequence_schema`	`sql_identifier`	包含序列的模式名称
`sequence_name`	`sql_identifier`	序列名称
`data_type`	`character_data`	序列的数据类型。在PostgreSQL中，当前总是 `bigint` 。
`numeric_precision`	`cardinal_number`	该字段（公开的或隐含的）包含序列数据类型的精度（见上述）。该精度表明了有效数字的位数。 它可以用在十进制或者二进制中，在字段 `numeric_precision_radix` 中说明。
`numeric_precision_radix`	`cardinal_number`	该字段表明该值在字段 `numeric_precision` 和 `numeric_scale` 中被表示。值为要么是2要么是10。
`numeric_scale`	`cardinal_number`	该字段（公开地或隐含地）包含序列数据类型的数值范围（见上文）。 数值范围表明了小数点右边有效数字的位数。它可以用在十进制或二进制中， 在字段 `numeric_precision_radix` 中说明。
`start_value`	`character_data`	序列的起始值
`minimum_value`	`character_data`	序列的最小值
`maximum_value`	`character_data`	序列的最大值
`increment`	`character_data`	序列的增量
`cycle_option`	`yes_or_no`	如果序列是周期的，那么是 `YES` ，否则是 `NO`

请注意，为了与SQL标准一致，起始值、最小值、最大值和增量值都作为字符串返回。

34.43. `sql_features`

表 `sql_features` 包含有关PostgreSQL 支持的在 SQL 标准里定义的正式特性的信息。这些信息和在Appendix D 出现的信息相同。你也可以在那里找到一些额外的背景信息。

Table 34-41. `sql_features` 字段

名字	数据类型	描述
`feature_id`	`character_data`	这个特性的标识字串
`feature_name`	`character_data`	这个特性的描述性名字
`sub_feature_id`	`character_data`	子特性的标识符字串，如果不是子特性，那么就是一个零长的字串
`sub_feature_name`	`character_data`	子特性的描述性名字，如果不是子特性，那么就是一个零长的字串
`is_supported`	`yes_or_no`	如果当前版本的PostgreSQL完全支持该特性，那么是 `YES` ，否则为 `NO`
`is_verified_by`	`character_data`	总是为空，因为PostgreSQL开发组并不做特性核实的正式测试。
`comments`	`character_data`	可能是一个有关特性支持状态的注释

34.44. `sql_implementation_info`

表 `sql_implementation_info` 包含有关 SQL 标准里各种留给具体实现定义的特性的信息。 这些信息主要用在 ODBC 接口的环境里；其它接口的用户可能会觉得这些信息没有什么用处。出于这个原因，独立的实现信息条目没有在这个描述；你会在 ODBC 接口的描述里找到它们。

Table 34-42. `sql_implementation_info` 字段

名字	数据类型	描述
`implementation_info_id`	`character_data`	实现信息条目的标识字串
`implementation_info_name`	`character_data`	实现信息条目的描述性名称
`integer_value`	`cardinal_number`	实现信息条目的数值，或如果数值在 `character_value` 字段里包含了，则为空。
`character_value`	`character_data`	实现信息条目的数值，或如果该值已经在字段 `integer_value` 里包含，则为空。
`comments`	`character_data`	可能是一个描述此实现信息条目的注释

34.45. `sql_languages`

表 `sql_languages` 为PostgreSQL 里支持的每个 SQL 语言绑定都包含一行。PostgreSQL 支持直接的 SQL 和在 C 里面嵌入 SQL； 这就是你从这个表里能看到的所有东西。

这个表在SQL:2008中已经从SQL标准中移除了，所有SQL:2003以后的标准都没有可以参考的条目了。

Table 34-43. `sql_languages` 字段

名字	数据类型	描述
`sql_language_source`	`character_data`	语言定义源的名称；总是 `ISO 9075`， 也就是SQL 标准
`sql_language_year`	`character_data`	`sql_language_source` 里引用的标准通过的年代。
`sql_language_conformance`	`character_data`	这种语言绑定的标准遵循级别。 对于 ISO 9075:2003, 这里总是 `CORE` 。
`sql_language_integrity`	`character_data`	总是空（这个数值与早期的SQL 标准相关。）
`sql_language_implementation`	`character_data`	总是为空
`sql_language_binding_style`	`character_data`	语言绑定风格， 要么是 `DIRECT` ， 要么是 `EMBEDDED` 。
`sql_language_programming_language`	`character_data`	如果绑定风格是 `EMBEDDED` ， 那么是编程语言， 否则是空，PostgreSQL只支持 C 语言。

34.46. `sql_packages`

表 `sql_packages` 包含有关定义在 SQL 标准里的那个特性包是PostgreSQL 支持的信息。请参考Appendix D获取有关特性包的背景信息。

Table 34-44. `sql_packages` 字段

名字	数据类型	描述
`feature_id`	`character_data`	包的标识字串
`feature_name`	`character_data`	包的描述性名字
`is_supported`	`yes_or_no`	如果该包被当前版本的PostgreSQL完全支持，则为 `YES` ， 否则为 `NO`
`is_verified_by`	`character_data`	总是空，因为PostgreSQL开发组没有进行任何特性兼容性的正式测试
`comments`	`character_data`	可能是一个有关该包支持状态的评注

34.47. `sql_parts`

表 `sql_parts` 包含关于PostgreSQL支持的SQL标准的几部分信息。

Table 34-45. `sql_parts` 字段

名字	数据类型	描述
`feature_id`	`character_data`	一个包含部分数量的标识字符串
`feature_name`	`character_data`	部分描述名称
`is_supported`	`yes_or_no`	如果当前PostgreSQL版本完全支持该部分为 `YES` ，否则为 `NO`
`is_verified_by`	`character_data`	总为空，因为PostgreSQL开发组不执行正规的特性一致性测试
`comments`	`character_data`	可能的一个关于支持部分状态的意见

34.48. `sql_sizing`

表 `sql_sizing` 包含有关PostgreSQL 里面各种大小限制和最大值的信息。这个信息特别用于在 ODBC 接口的环境下；其它接口的用户很可能发现这个信息没什么用。处于这个原因，在这里没有描述独立的大小条目；你将在 ODBC 接口的描述里找到它们。

Table 34-46. `sql_sizing` 字段

名字	数据类型	描述
`sizing_id`	`cardinal_number`	尺寸项的标识符
`sizing_name`	`character_data`	尺寸项的描述名称
`supported_value`	`cardinal_number`	尺寸项的数值，如果尺寸无限制或者无法确定，则为 0，如果不支持适用的尺寸项的特性，则为空。
`comments`	`character_data`	可能是一个有关此尺寸项的评注

34.49. `sql_sizing_profiles`

表 `sql_sizing_profiles` 包含有关 SQL 标准要求的各种配置文件的 `sql_sizing` 信息。PostgreSQL并不跟踪任何 SQL 配置文件，所以这个表是空的。

Table 34-47. `sql_sizing_profiles` 字段

名字	数据类型	描述
`sizing_id`	`cardinal_number`	尺寸项的标识符
`sizing_name`	`character_data`	尺寸项的描述性名称
`profile_id`	`character_data`	一个配置文件的标识字串
`required_value`	`cardinal_number`	SQL 配置文件对尺寸项要求的数值，如果配置文件对尺寸项没有限制，则为0， 如果配置文件对尺寸项所适用的特性没有任何要求，那么就为空。
`comments`	`character_data`	可能是一个有关该配置文件里面的尺寸项的注释

34.50. `table_constraints`

视图 `table_constraints` 包含所有属于当前用户的表里面的约束， 或有除了 `SELECT` 之外的某些权限。

Table 34-48. `table_constraints` 字段

名字	数据类型	描述
`constraint_catalog`	`sql_identifier`	包含该约束地数据库的名称（总是当前数据库）
`constraint_schema`	`sql_identifier`	包含这个约束的模式名称
`constraint_name`	`sql_identifier`	约束名称
`table_catalog`	`sql_identifier`	包含该表的数据库名称（总是当前数据库）
`table_schema`	`sql_identifier`	包含此表的模式名
`table_name`	`sql_identifier`	表名
`constraint_type`	`character_data`	约束的类型： `CHECK` , `FOREIGN KEY` , `PRIMARY KEY` , 或者 `UNIQUE`
`is_deferrable`	`yes_or_no`	如果约束可以推迟，为 `YES` ，否则为 `NO` 。
`initially_deferred`	`yes_or_no`	如果约束是可以推迟的并且是首先推迟的，为 `YES` ，否则为 `NO` 。

34.51. `table_privileges`

视图 `table_privileges` 标识所有赋与当前用户或者由当前用户赋予的， 在表或者视图上的权限。每个表、赋权人、受权人的组合都有一行。

Table 34-49. `table_privileges` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	授予该权限的角色名
`grantee`	`sql_identifier`	被授予该权限的角色名
`table_catalog`	`sql_identifier`	包含该表的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	包含该表的模式名
`table_name`	`sql_identifier`	表名
`privilege_type`	`character_data`	权限的类型： `SELECT` ， `INSERT` ， `UPDATE` ， `DELETE` ， `TRUNCATE` ， `REFERENCES` ，或 `TRIGGER`
`is_grantable`	`yes_or_no`	如果权限是可赋予的，则为 `YES` ，否则为 `NO`
`with_hierarchy`	`yes_or_no`	在SQL标准中， `WITH HIERARCHY OPTION` 是一个单独的（子）特权， 允许在表继承层次结构上执行创建。在PostgreSQL中，这包含在 `SELECT` 特权中，所以如果特权是 `SELECT` 这个字段就显示 `YES` ， 否则为 `NO` 。

34.52. `tables`

视图 `tables` 包含所有在当前数据库里定义的表和视图。 只有那些当前用户有权限访问（要么是所有者，要么是有某些权限）的表和视图才显示出来。

Table 34-50. `tables` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含该表的数据库名（总是当前数据库）
`table_schema`	`sql_identifier`	包含该表的模式名
`table_name`	`sql_identifier`	表名
`table_type`	`character_data`	表的类型：永久基本表是 `BASE TABLE` （普通的表类型）， 视图是 `VIEW` ， 外表是 `FOREIGN TABLE` ， 或者是 `LOCAL TEMPORARY` 表示临时表。
`self_referencing_column_name`	`sql_identifier`	应用于一个PostgreSQL里没有的特性
`reference_generation`	`character_data`	应用于一个PostgreSQL里没有的特性
`user_defined_type_catalog`	`sql_identifier`	如果该表是一个指定类型的表，那么就是包含基本的数据类型的数据库名称 （总是当前数据库）， 否则为空。
`user_defined_type_schema`	`sql_identifier`	如果该表是一个指定类型的表，那么就是包含基本的数据类型的模式名称，否则为空。
`user_defined_type_name`	`sql_identifier`	如果该表是一个指定类型的表，那么为基础数据类型的名称，否则为空。
`is_insertable_into`	`yes_or_no`	如果该表可插入则为 `YES` ， 否则为 `NO` （基础表总是可插入的，视图不是必须的。）
`is_typed`	`yes_or_no`	如果该表是一个指定类型的表则为 `YES` ， 否则为 `NO`
`commit_action`	`character_data`	目前未实现

34.53. `triggered_update_columns`

对于当前数据库中的触发器来说指定一个字段列表（像 `UPDATE OF column1, column2` ）， 视图 `triggered_update_columns` 识别这些字段。触发器不指定一个不包含在这个视图中的字段列表。 只有那些当前用户拥有或有除了 `SELECT` 之外的某些权限的字段才显示。

Table 34-51. `triggered_update_columns` 字段

名字	数据类型	描述
`trigger_catalog`	`sql_identifier`	包含该触发器的数据库名称（总是当前数据库）
`trigger_schema`	`sql_identifier`	包含该触发器的模式名称
`trigger_name`	`sql_identifier`	触发器的名称
`event_object_catalog`	`sql_identifier`	包含定义了触发器的表的数据库名称（总是当前数据库）
`event_object_schema`	`sql_identifier`	包含定义了触发器的表的模式名称
`event_object_table`	`sql_identifier`	定义了触发器的表名
`event_object_column`	`sql_identifier`	定义了触发器的字段的名称

34.54. `triggers`

视图 `triggers` 包含了所有在当前数据库中的表和视图上定义的， 并且当前用户是其所有者或有除了 `SELECT` 之外的某些权限的触发器。

Table 34-52. `triggers` 字段

名字	数据类型	描述
trigger_catalog	sql_identifier	包含该触发器的数据库名称（总是当前数据库）
trigger_schema	sql_identifier	包含该触发器的模式名称
trigger_name	sql_identifier	触发器名称
event_manipulation	character_data	激发触发器的事件（ INSERT ， UPDATE 或者 DELETE ）
event_object_catalog	sql_identifier	包含触发器定义所在表的数据库名称（总是当前数据库）
event_object_schema	sql_identifier	包含触发器定义所在表的模式名
event_object_table	sql_identifier	触发器定义所在的表名
action_order	cardinal_number	尚未实现
action_condition	character_data	WHEN 触发器条件，如果没有则为空 （如果当前角色不是该表的所有者也为空）
action_statement	character_data	触发器执行的语句（目前总是 EXECUTE PROCEDURE _function_ (...))
action_orientation	character_data	标识触发器是对处理的每一行激发还是对每个语句激发 （ ROW 或者 STATEMENT ）
action_timing	character_data	触发器触发的时间（ BEFORE ， AFTER 或者 INSTEAD OF ）
action_reference_old_table	sql_identifier	应用于一个PostgreSQL里没有的特性
action_reference_new_table	sql_identifier	应用于一个PostgreSQL里没有的特性
action_reference_old_row	sql_identifier	应用于一个PostgreSQL里没有的特性
action_reference_new_row	sql_identifier	应用于一个PostgreSQL里没有的特性
created	time_stamp	应用于一个PostgreSQL里没有的特性

PostgreSQL里面的触发器在影响到信息模式的表现形式方面， 与 SQL 标准有两处不同。首先，在PostgreSQL里，触发器名字是表示本地的对象， 而不是独立的模式对象。因此，我们可以在一个模式里定义重复的触发器名字，只要他们属于不同的表。（ trigger_catalog 和 trigger_schema 实际上是属于触发器定义所在表的值。） 第二，

PostgreSQL里的触发器可以定义为在多个事件上触发 （比如`ON INSERT OR UPDATE`），而SQL 标准只允许一个。 如果一个触发器定义为在多个事件上触发，那么在信息模式里它会表现为多行，事件的每个类型一行。因为这两个原因， 视图`triggers`的主键实际上是`(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)`，而不是`(trigger_catalog, trigger_schema, trigger_name)`， 后者是 SQL 标准声明的。当然，如果你定义一个遵循 SQL 标准的触发器（触发器名字在模式中唯一， 并且每个触发器只有一个事件类型），这些事情不会烦着你。

> **Note:** 在PostgreSQL 9.1之前，这个视图的字段`action_timing`, `action_reference_old_table`, `action_reference_new_table`, `action_reference_old_row`, 和 `action_reference_new_row`分别叫做`condition_timing`, `condition_reference_old_table`, `condition_reference_new_table`, `condition_reference_old_row`, 和 `condition_reference_new_row`。 这是它们在SQL:1999标准中的名字。新的名字符合SQL:2003和之后的标准。

34.55. `udt_privileges`

视图 `udt_privileges` 标识授予当前用户或由当前用户授予用户定义类型的 `USAGE` 权限。对于每个字段、授权者和受权者的组合都有一行。 这个视图只显示复合类型（参阅Section 34.57获得原因）；参阅Section 34.56获取域的权限。

Table 34-53. `udt_privileges` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	授予特权的角色名称
`grantee`	`sql_identifier`	被授予特权的角色名称
`udt_catalog`	`sql_identifier`	包含该类型的数据库名称（总是当前数据库）
`udt_schema`	`sql_identifier`	包含该类型的模式名称
`udt_name`	`sql_identifier`	类型名
`privilege_type`	`character_data`	总是 `TYPE USAGE`
`is_grantable`	`yes_or_no`	如果权限可授予，则为 `YES` ，否则为 `NO`

34.56. `usage_privileges`

视图 `usage_privileges` 用于标识在各种类型的对象上赋与当前用户或者当前用户赋与的 `USAGE` 权限。在 PostgreSQL中， 当前适用于排序规则、域、外部数据封装、外部服务器和序列。每个对象、授权者和受权者的组合都有一行。

因为在PostgreSQL里，排序规则并没有真正的权限， 所以这个视图显示了隐含的由 `PUBLIC` 的所有者授予所有排序规则的非可授予的 `USAGE` 权限。其他的对象类型显示了真正的权限。

在PostgreSQL中，序列也支持 `SELECT` 和 `UPDATE` 权限， 除了 `USAGE` 权限。这不是标准并且因此在信息模式中不可见。

Table 34-54. `usage_privileges` 字段

名字	数据类型	描述
`grantor`	`sql_identifier`	授权的角色名
`grantee`	`sql_identifier`	被授权的角色名
`object_catalog`	`sql_identifier`	包含该对象的数据库名（总是当前数据库）
`object_schema`	`sql_identifier`	如果适用，是包含该对象的模式名，否则是空字符串
`object_name`	`sql_identifier`	对象名
`object_type`	`character_data`	`COLLATION` 或 `DOMAIN` 或 `FOREIGN DATA WRAPPER` 或 `FOREIGN SERVER` 或 `SEQUENCE`
`privilege_type`	`character_data`	总是 `USAGE`
`is_grantable`	`yes_or_no`	如果权限可授予则为 `YES` ，否则为 `NO`

34.57. `user_defined_types`

视图 `user_defined_types` 当前包含所有在当前数据库中定义的复合类型。 只有当前用户可以访问的类型才显示出来（通过成为其所有者或有某些权限）。

SQL知道两种用户定义的类型：结构化类型（在PostgreSQL中也成为复合类型） 和不同类型（在PostgreSQL中没有实现）。为了永不过时， 使用字段 `user_defined_type_category` 来区分它们。其他用户定义类型， 例如基础类型和枚举类型，是PostgreSQL的扩展，不在这里展示。 对于域，参阅Section 34.22。

Table 34-55. `user_defined_types` 字段

名字	数据类型	描述
`user_defined_type_catalog`	`sql_identifier`	包含该类型的数据库名称（总是当前数据库）
`user_defined_type_schema`	`sql_identifier`	包含该类型的模式名称
`user_defined_type_name`	`sql_identifier`	类型名
`user_defined_type_category`	`character_data`	当前总是 `STRUCTURED`
`is_instantiable`	`yes_or_no`	适用于一个PostgreSQL中没有的特性
`is_final`	`yes_or_no`	适用于一个PostgreSQL中没有的特性
`ordering_form`	`character_data`	适用于一个PostgreSQL中没有的特性
`ordering_category`	`character_data`	适用于一个PostgreSQL中没有的特性
`ordering_routine_catalog`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`ordering_routine_schema`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`ordering_routine_name`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`reference_type`	`character_data`	适用于一个PostgreSQL中没有的特性
`data_type`	`character_data`	适用于一个PostgreSQL中没有的特性
`character_maximum_length`	`cardinal_number`	适用于一个PostgreSQL中没有的特性

`character_octet_length`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`character_set_catalog`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`character_set_schema`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`character_set_name`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`collation_catalog`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`collation_schema`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`collation_name`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`numeric_precision`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`numeric_precision_radix`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`numeric_scale`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`datetime_precision`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`interval_type`	`character_data`	适用于一个PostgreSQL中没有的特性
`interval_precision`	`cardinal_number`	适用于一个PostgreSQL中没有的特性
`source_dtd_identifier`	`sql_identifier`	适用于一个PostgreSQL中没有的特性
`ref_dtd_identifier`	`sql_identifier`	适用于一个PostgreSQL中没有的特性

34.58. `user_mapping_options`

视图 `user_mapping_options` 包含在当前数据库中为用户映射定义的所有项。 只有那些当前用户可以访问相应外部服务器的用户映射才显示（通过成为其所有者或有某些权限）。

Table 34-56. `user_mapping_options` 字段

名字	数据类型	描述
`authorization_identifier`	`sql_identifier`	被映射的用户名称，或如果映射是公共的为 `PUBLIC`
`foreign_server_catalog`	`sql_identifier`	该映射使用的外部服务器定义所在的数据库名称（总是当前数据库）
`foreign_server_name`	`sql_identifier`	该映射使用的外部服务器的名称
`option_name`	`sql_identifier`	选项名
`option_value`	`character_data`	选项值。这个字段将显示null除非当前用户是被映射的用户， 或映射是 `PUBLIC` 并且当前用户是服务器拥有者， 或当前用户是超级用户。这样的目的是为了保护密码信息作为用户映射选项存储。

34.59. `user_mappings`

视图 `user_mappings` 包含所有定义在当前数据库中的用户映射。 只有那些当前用户可以访问相应的外部服务器的用户映射才显示（通过成为所有者或拥有一些特权）。

Table 34-57. `user_mappings` 字段

名字	数据类型	描述
`authorization_identifier`	`sql_identifier`	被映射的用户名称，或如果映射是公共的为 `PUBLIC`
`foreign_server_catalog`	`sql_identifier`	被该映射使用的外部服务器定义所在的数据库名称（总是当前数据库）
`foreign_server_name`	`sql_identifier`	被该映射使用的外部服务器的名称

34.60. `view_column_usage`

视图 `view_column_usage` 标识所有在一个视图的查询表达式 （定义视图的 `SELECT` 语句）中使用的字段。 只有在当前用户是包含该字段的表的所有者的时候才会列出这个字段。

> **Note:** 系统表的字段没有列出。以后应该修补这个问题。

Table 34-58. `view_column_usage` 字段

名字	数据类型	描述
`view_catalog`	`sql_identifier`	包含这个视图的数据库名称（总是当前数据库）
`view_schema`	`sql_identifier`	包含这个视图的模式名称
`view_name`	`sql_identifier`	视图名称
`table_catalog`	`sql_identifier`	含被这个视图使用的字段的表所在的数据库的名字（总是当前数据库）
`table_schema`	`sql_identifier`	含被这个视图使用的字段的表所在的数据库的名字（总是当前数据库）
`table_name`	`sql_identifier`	包含被这个视图使用的字段的表名
`column_name`	`sql_identifier`	视图使用的字段名

34.61. `view_routine_usage`

视图 `view_routine_usage` 识别所有被应用在视图 的查询表达式（定义该视图的 `SELECT` 语句）中的日常活动（函数和程序）。 只有当前用户是这个活动的所有者的时候才包含这个活动。

Table 34-59. `view_routine_usage` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含该视图的数据库的名称（总是当前数据库）
`table_schema`	`sql_identifier`	包含该视图的模式的名称
`table_name`	`sql_identifier`	视图名
`specific_catalog`	`sql_identifier`	包含该函数的数据库的名称（总是当前数据库）
`specific_schema`	`sql_identifier`	包含该函数的模式的名称
`specific_name`	`sql_identifier`	函数的"专用名"。参阅Section 34.40获取更多信息。

34.62. `view_table_usage`

视图 `view_table_usage` 标识所有在一个视图的查询表达式中 （定义视图的 `SELECT` 语句）使用的表的名字。 只有在当前用户是这个表的所有者的时候才会包含这个表。

> **Note:** 没有包括系统表。这个在将来应该修补。

Table 34-60. `view_table_usage` 字段

名字	数据类型	描述
`view_catalog`	`sql_identifier`	包含该视图的数据库的名称（总是当前数据库）
`view_schema`	`sql_identifier`	包含该视图的模式的名称
`view_name`	`sql_identifier`	视图名
`table_catalog`	`sql_identifier`	包含被视图使用的表的数据库的名称（总是当前数据库）
`table_schema`	`sql_identifier`	包含被视图使用的表的模式名称
`table_name`	`sql_identifier`	视图使用的表的名称

34.63. `views`

视图 `views` 包含所有定义在当前数据库中的视图。 只有那些当前用户有权访问（要么是所有者要么是拥有某些权限）的视图才显示出来。

Table 34-61. `views` 字段

名字	数据类型	描述
`table_catalog`	`sql_identifier`	包含这个视图的数据库名称（总是当前数据库）
`table_schema`	`sql_identifier`	包含这个视图的模式名称
`table_name`	`sql_identifier`	视图名称
`view_definition`	`character_data`	定义视图的查询表达式（如果当前用户不是视图所有者，则为空）
`check_option`	`character_data`	应用于一个PostgreSQL里没有的特性
`is_updatable`	`yes_or_no`	如果视图是可更新的则为 `YES` （允许 `UPDATE` 和 `DELETE`)），否则为 `NO`
`is_insertable_into`	`yes_or_no`	如果视图是可插入的则为 `YES` （允许 `INSERT` ），否则为 `NO`
`is_trigger_updatable`	`yes_or_no`	如果在视图上定义了一个 `INSTEAD OF` `UPDATE` 触发器则为 `YES` ，否则为 `NO`
`is_trigger_deletable`	`yes_or_no`	如果在视图上定义了一个 `INSTEAD OF` `DELETE` 触发器则为 `YES` ，否则为 `NO`
`is_trigger_insertable_into`	`yes_or_no`	如果在视图上定义了一个 `INSTEAD OF` `INSERT` 触发器则为 `YES` ，否则为 `NO`

V. 服务器端编程

这部分是关于用户怎样通过增加用户定义的类型、操作符、聚集、查询语言和编程语言函数来扩展服务器功能。 这些是高级主题，可能需要在阅读完并理解所有其它 PostgreSQL 手册之后才能阅读。这部分后面的章节还描述了在PostgreSQL 里的服务器端编程语言。我们至少要读过Chapter 35的头几节才能深入阅读有关服务器端编程语言的材料。

Table of Contents

Chapter 35. 扩展SQL

Table of Contents

在本章的剩余部分， 我们将讨论你如何通过增加下面几种对象来扩展PostgreSQL 的SQL查询语言：

- 函数（开始于Section 35.3）
- 聚集（开始于Section 35.10）
- 数据类型（开始于Section 35.11）
- 操作符（开始于Section 35.12）
- 索引操作符类（开始于Section 35.14）
- 相关对象打包（开始于Section 35.15）

35.1. 扩展性是如何实现的

PostgreSQL可扩展的原因是它的操作是由表驱动的。 如果你熟悉标准的关系数据库系统，你就知道它们把数据库、表、字段等信息存储在一个被称为系统表 (有些系统称为数据字典) 的地方。这些表与其它表没什么不同， 只不过DBMS把它自己内部的信息存放于此罢了。PostgreSQL与其它系统的不同之处在于它在系统表里存储了更多的信息： 除了关于表和列/字段的信息之外， 还有关于它们的类型、函数、访问方式之类的信息。这些表可以被用户修改， 而且由于PostgreSQL的内部操作是以这些表为基础的， 这就意味着PostgreSQL可以被用户进行扩展。相比之下， 传统的数据库系统只能通过修改源代码或加载由DBMS供应商提供的特殊模块来扩展。

PostgreSQL还可以通过动态加载的方法与用户书写的代码结合在一起。 也就是说，用户可以把一个目标代码文件(通常是共享库) 声明为一个新类型或函数的实现， 这时PostgreSQL将根据需要加载它们。 用SQL书写的代码甚至更容易加入到服务器中去。 这种可以"在线"更改操作的能力 使PostgreSQL特别适合于新应用和新存储结构的快速定型。

35.2. PostgreSQL类型系统

PostgreSQL数据类型可以分为基本类型、复合类型、域、伪类型。

35.2.1. 基本类型

基本类型是那些在SQL语言层次更低级别(通常是C语言)上实现的类型(比如 `int4` 类型)， 它们通常与抽像数据类型对应。PostgreSQL对这些数据类型只能通过用户提供的函数来操作，并且对这些数据类型行为的理解只限于用户所描述的范围。基本类型进一步分成标量和数组类型。 对于每种标量类型，系统都会自动创建一个对应的数组类型，可以保存该标量类型的变长数组。

35.2.2. 复合类型

复合类型(或者说行类型)是用户创建表时创建的。 也可以用CREATE TYPE创建一个"独立的"、没有关联表的复合类型。 复合类型只是一个带着相关字段名称的基本类型的列表。 复合类型的数值是一行字段值或者一条字段值组成的记录。 用户可以从SQL查询里访问其字段。参考Section 8.16获取更多复合类型的相关信息。

35.2.3. 域

域基于一种特定的基本类型，从很多角度来看，它们也可以和其对应的基本类型交换。 但是，域可以有约束，把它的有效值限制在其对应的基本类型的有效值范围的一个子集中。

域可以使用SQL命令CREATE DOMAIN创建。 它们的创建和使用不在本章讨论。

35.2.4. 伪-类型

有一些用于特殊目的的"伪类型"。伪类型不能作为表的字段类型， 也不能作为复合类型的属性，但是它们可以用于声明函数的参数和结果类型。 这样就在类型系统里提供了一个标识特殊类型函数的机制。 Table 8-24列出了现有的伪类型。

35.2.5. 多态类型

`anyelement`, `anyarray`, `anynonarray`, `anyenum`, 和 `anyrange` 是五种特别有趣的伪类型，它们被称作多态类型。 任何用这些类型定义的函数就叫做多态函数。 一种多态函数可以在许多不同的数据类型上操作， 它们根据调用中实际传递进来的数据类型判断具体的类型。

多态参数和结果是相互绑定的，并且在分析查询调用的函数时解析成特定的数据类型。 每个声明成 `anyelement` 的位置(参数或者返回类型)都允许拥有一个特定的实际数据类型， 但是在任何给定的调用过程中，它们都必须是同样的类型。 每个声明为 `anyarray` 的位置都可以是任何数组数据类型， 类似的，声明为 `anyrange` 的位置 也必许都是同样的类型。而且，如果有些位置声明为 `anyarray` 而其它位置声明为 `anyelement` ， 那么在 `anyarray` 位置上的类型必须是元素类型与那些出现在 `anyelement` 位置上的类型相同的数组。 类似的，如果有声明为 `anyrange` 的位置而且其他的声明为 `anyelement` ，那么 在 `anyrange` 位置上的类型必须是子类型与那些出现在 `anyelement` 位置上类型相同 的范围。 `anynonarray` 实际上被看做 `anyelement` ， 但却多一个约束：实际类型必须不能是一个数组类型。 `anyenum` 实际上被看做 `anyelement` ，但却多一个约束： 实际类型必须是一个枚举类型。

因此，如果多个参数位置声明为多态类型，其实际效果是只允许某些实际参数类型的组合出现。 比如，一个函数声明为 `equal(anyelement, anyelement)` 将接受任何两个输入值，只要它们的数据类型相同。

如果一个函数的返回值声明为多态类型，那么至少有一个参数位置也是多态的， 并且提供给参数的类型决定本次调用实际返回的类型。比如，如果没有数组下标机制， 那么我们可以定义一个实现下标的函数 `subscript(anyarray, integer) returns anyelement` 。 这个声明约束第一个实际参数是一个数组类型，并且允许分析器从第一个参数的实际类型推导出正确的返回类型。 声明为一个 `f(anyarray) returns anyenum` 的函数的另一个例子只接受枚举类型的数组。

需要注意的是， `anynonarray` 和 `anyenum` 不代表不同的类型变量； 它们是与 `anyelement` 相同的类型，只有一个额外的约束。 例如，声明一个函数为 `f(anyelement, anyenum)` 等同于声明它为 `f(anyenum, anyenum)` ：两个实际参数必须是相同的枚举类型。

一个可变参数函数（其使用一个可变数目的参数，如Section 35.4.5中描述） 可以是多态的： 可以通过声明它的最后一个参数为 `VARIADIC` `anyarray` 来实现。 为了实现参数匹配并决定实际结果类型， 这样一个函数的行为等同于将 `anynonarray` 参数写一个合适的数目。

35.3. 用户定义的函数

PostgreSQL提供了四种函数：

- 查询语言函数（函数缩写SQL）(Section 35.4)
- 程序语言函数 (函数缩写,比如，PL/pgSQL或者PL/Tcl) (Section 35.7)
- 内部函数(Section 35.8)
- C-语言函数(Section 35.9)

每一种函数可以采用基本类型，复合类型，或者两者的组合作为参数。 另外，每种函数可以返回基本类型或者复合类型。函数也可以定义为返回基本或者复合值的集合。

许多种函数可以接受或者返回某些伪类型（比如多态类型），但是可用设施不同。 查阅各种函数的描述以获取更多详细信息。

最容易定义SQL函数，因此我们将开始讨论这些。 大多数用于SQL函数的概念将和其它类型的函数一致。

在本章中，参考CREATE FUNCTION命令手册页对于更好的理解例子是很有帮助的。 本章的例子还可以在PostgreSQL源码发布的 `src/tutorial` 目录的 `funcs.sql` 和 `funcs.c` 中找到。

35.4. 查询语言(SQL)函数

SQL函数执行SQL语句的任意列表，返回列表中最后一个查询结果。 在简单情况下（非-集合），将返回最后查询结果的第一行。 （记住多行结果的"第一行"是不明确的 除非你使用 `ORDER BY` 。） 如果最后查询没有返回任何行，则返回空值。

另外，一个SQL函数可以声明为返回一个集合（即多行）。 方法是把该函数的返回类型声明为 `SETOF` `_sometype_` 。 或者等价声明它为 `RETURNS TABLE(``_columns_`)。 这种情况下，最后一条查询结果的所有行都会被返回。更多细节在下面讲解。

SQL函数的函数体应该是一个用分号分隔的SQL语句列表。 最后一个语句后面的分号是可选的。除非函数声明为返回 `void` ， 否则最后一条语句必须是 `SELECT` 或者 `INSERT` ， `UPDATE` 或者有 `RETURNING` 子句的 `DELETE` 。

任何SQL命令集合都可以打包在一起，定义成新的函数。 除了 `SELECT` 查询之外，命令可以包含修改数据的查询 (`INSERT` ， `UPDATE` 和 `DELETE`)以及其它 SQL命令。 （你不能使用事务控制命令，比如 `COMMIT` ， `SAVEPOINT` 和 一些实用命令，比如 `VACUUM` ， SQL）。 不过，最后一条命令必须是一个 `SELECT` 语句， 或者有 `RETURNING` 子句返回函数的返回类型。 另外， 如果你只想定义一连串动作而无需返回任何数值，可以定义返回 `void` 。 比如， 下面这个函数从 `emp` 表删除负数的薪水：

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

`CREATE FUNCTION` 命令的语法要求函数体写成一个字符串文本。 一般来说，字符串常量使用美元符界定更方便些(参阅Section 4.1.2.4)。 如果你决定使用通常的字符串常量语法，你必须加单引号标记(`'`)和和反斜杠(`\`)， 在函数体中（假定使用逃逸字符串语法） （参见Section 4.1.2.1）。

35.4.1. Arguments for SQL Functions

在函数体中使用名称或数字引用SQL函数的参数。这两种方法的例子在下面。

使用一个名字，声明有名称的函数参数，然后在函数体中写上这个名字。如果参数名称 在当前SQL命令的同一函数中与任何列的名称相同，将优先考虑列名称。为了重写， 限定参数名与函数名本身，也就是说 `_function_name_` . `_argument_name_` 。（如果有一个合格的列名称冲突，再次列名称获胜。你可以通过选择一个SQL命令表不同的别名来避免歧义。）

在旧的数值方法中，使用语法 `$``_n_` : `$1` 引用第一个输入参数， `$2` 到第二个，等等。是否声明带有名字的特定参数将要工作。

如果一个参数是复合类型，然后圆点标记法，比如， `argname.fieldname` 或者 `$1.fieldname` 可以用于访问参数属性。再次，你可能需要限定函数名的参数名 来形成模糊参数名形式。

SQL函数参数只能作为数据值使用，而不能作为标示符。因此比如这是合理的：

```
INSERT INTO mytable VALUES ($1);
```

but this will not work:

```
INSERT INTO $1 VALUES (42);
```

> **Note:** 使用名称引用SQL函数参数的功能被添加到PostgreSQL 9.2中。 在旧的服务器中使用的函数必须使用 `$``_n_` 标记法。

35.4.2. 基本类型上的SQL函数

最简单的SQL函数可能没有参数并且返回一个基本类型， 比如一个返回 `integer` 的函数：

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;
   --另外一种字符串文本的语法：
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

请注意我们在函数体里面定义了一个字段别名(`result`)用于函数结果， 但是这个字段别名在函数外面是不可见的。因此， 结果是以 `one` 而不是 `result` 为标签的。

定义一个接受基本类型做参数的SQL函数几乎一样简单。

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

或者，我们可以摒弃参数名，并且使用数字：

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

下面是一个更有用的函数，我们可以用它对一个银行帐号做扣款动作：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

可以像下面这样用这个函数给帐户17扣款$100.00：

```
SELECT tf1(17, 100.0);
```

在这个例子中，我们选择名称 `accountno` 作为第一个参数，但是这和 `bank` 表中的列名是一样的。在 `UPDATE` 命令中， `accountno` 引用列 `bank.accountno` ， 因此，必须使用 `tf1.accountno` 来引用参数。当然我们可以通过使用 参数的不同名称来避免这种情况。

实际上我们可能希望函数有一个比常量1更有用一些的结果。所以实用的定义可能是

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

它修改余额并返回新的余额。可以在命令中使用 `RETURNING` 做同样的事情：

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.3. 复合类型上的SQL函数

当书写使用用复合类型做参数的函数时，不仅要声明需要哪个参数， 而且要声明参数的字段(数据域)。比如，假设 `emp` 是一个包含雇员信息的表， 并且因此也是该表每行的复合类型的名字。一个计算某人薪水翻番之后数值的 `double_salary` 函数：

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

请注意这里使用 `$1.salary` 语法选择参数行数值的一个字段。 还要注意 `SELECT` 命令使用 `*` 表示该表的整个当前行作为复合数值。 表里面的行也可以用表名字引用， 像下面这样：

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

不过这个用法已经废弃了，因为很容易导致混淆。

有时候用 `ROW` 构造器动态地构造一个复合参数值也很有用。 比如，我们可以调节传递给函数的数据：

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

也可以写一个返回复合类型的函数。下面是一个只返回一行的 `emp` 函数：

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

在这个例子中我们给每个字段都赋予了一个常量， 当然也可以用任何表达式来代替这些常量。

注意定义函数的两个重要问题:

- 选择列表的顺序必须和与该复合类型相关的表中字段的顺序完全一样。 像上面那样给字段命名是和系统毫无关系的。
- 你必须对表达式进行类型转换以匹配复合类型的定义。 否则你将看到下面的错误信息：

```
&lt;samp class="literal"&gt;ERROR:  function declared to return emp returns varchar
```

另外一个定义同样函数的方法是:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

这里的 `SELECT` 只返回对应复合类型的一个单独字段。 在这种情况下，这么做并没有任何好处，但是它在某些场合是一个很好用的东西— 比如，需要通过调用另外一个返回所需复合类型数值的函数来计算结果。

我们可以用任何两种方式直接调用这个函数：

```
SELECT new_emp();

         new_emp
--------------------------
 (None,1000.0,25,"(2,2)")

SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

第二种方法在Section 35.4.7里有更完整的描述。

在使用一个返回复合类型的函数时，你可以用下面的语法从结果中只抽取一个字段：

```
SELECT (new_emp()).name;

 name
------
 None
```

必须用一对额外的圆括弧防止分析器误解。 如果省略这对括弧就会看见类似下面这样的东西：

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

另外一个选择是使用函数表示法抽取字段。 解释这些问题的简单方法是交互使用 `attribute(table)` 和 `table.attribute` 表示法。

```
SELECT name(new_emp());

 name
------
 None
```

```
--上述语句与下面的这个相同：
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30;

SELECT name(emp) AS youngster FROM emp WHERE age(emp) < 30;

 youngster
-----------
 Sam
 Andy
```

> **Tip:** 函数表示法和字段属性表示法之间的等效关系让我们可以使用复合类型上的函数来模拟 "计算得出的字段"。 比如，使用前面的 `double_salary(emp)` 定义，我们可以写
>
> ```
> SELECT emp.name, emp.double_salary FROM emp;
> ```
>
> 应用可以直接这么使用而无需明确知道 `double_salary` 并不是表中一个真实的字段。 同样也可以模拟视图上计算出的字段。
>
> 因为这种操作，给函数采取单一复合类型参数与复合类型的任何字段名相同是不明智的。

还有一个使用函数返回复合类型的情况是把结果传递给另外一个输入该行类型的函数：

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

还可以把返回复合类型的函数当作一个表函数使用，如Section 35.4.7所述。

35.4.4. 带输出参数的SQL函数

描述函数的结果的另外一种方法是把它定义成带有输出参数的函数，比如：

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

这个版本和Section 35.4.2里面 的那个 `add_em` 版本没有什么本质的区别。 输出参数的真正价值在于它提供了定义返回多个字段的函数的便利方法。比如，

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

这里实际发生的事情是我们为函数的结果创建了一个匿名的复合类型。 上面的例子和下面的例子有同样的最终结果

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

不过，不用操心独立的复合类型定义通常都会很方便。请注意附属于输出参数的名称不仅仅是修饰，但 也决定了匿名复合类型的列名。（如果你为输出参数而忽略了名称，则系统将选择一个自己的名字）。

请注意，从SQL里调用这些函数的时候，输出参数并未包含在调用参数列表里。 这是因为PostgreSQL认为只有输入参数定义函数的调用签名。 这也意味着在类似删除函数这样的场合里， 只有输入参数管用。我们可以用下列命令之一删除上述函数

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

参数可以被标记为 `IN` (缺省), `OUT` , `INOUT` 或者 `VARIADIC` 。 `INOUT` 参数同时作为输入参数(调用参数列表的一部分) 和输出参数(结果记录类型的一部分)。 `VARIADIC` 参数是输入参数，但是作为描述文本特殊对待。

35.4.5. 带有参数可变数量的SQL

SQL函数声明接受参数可变数量，只要所有"optional" 参数有相同数据类型。可选参数将被作为数组传递给函数。函数通过把最后参数作为 `VARIADIC` 声明；这个参数必须声明为数组类型。比如：

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

实际上，达到或者超过 `VARIADIC` 位置的所有实际参数都被聚集为一维阵列，正如你写的

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

你可以不写，至少它不匹配这个函数定义。标记 `VARIADIC` 的参数匹配一个或多个元素类型 的发生，而不是固有类型。

有时候可以将已构建数组传递给可变参数函数；

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

这防止函数的可变参数扩展到它的元素类型，从而使数组参数值正常匹配。 `VARIADIC` 只可以附属于函数调用的最后一个实参。

数组元素的参数产生一个可变的参数作为没有自己的名字看待。这意味着它是不 可以使用命名参数(Section 4.3)调用一个可变参数函数， 除非你指定 `VARIADIC` 。例如，这项工作：

```
SELECT mleast(VARIADIC arr := ARRAY[10, -1, 5, 4.4]);
```

但不是这些：

```
SELECT mleast(arr := 10);
SELECT mleast(arr := ARRAY[10, -1, 5, 4.4]);
```

35.4.6. 具有参数缺省值的SQL函数

函数可以为了部分或全部输入参数而声明默认值。 当函数使用不充分的许多实际参数调用函数的时候，插入缺省值。因为参数 只能从实际的参数列表的末尾省略，所有 具有默认值的参数都有默认值。（虽然使用命名参数符号 可以让这个限制宽松，它仍然是强制的，位置参数符号合理运行。）

比如：

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- fails since there is no default for the first argument
ERROR:  function foo() does not exist
```

`=` 符号也可以用在关键字 `DEFAULT` 的位置。

35.4.7. 作为表数据源的SQL函数

所有SQL函数都可以在查询的 `FROM` 子句里使用。 但是它对于返回复合类型的函数特别有用。如果该函数定义为返回一个基本类型， 那么表函数生成一个单字段表。如果该函数定义为返回一个复合类型， 那么该表函数生成一个该复合类型里每个属性组成的行。

这里是一个例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

正如这个例子显示的那样，可以像对待一个普通表的字段一样对待函数的结果字段。

请注意我们只从该函数中获取了一行。这是因为没有使用 `SETOF` 。'这个问题在下一节讲述。

35.4.8. 返回集合的SQL函数

如果一个SQL函数声明为返回 `SETOF` `_sometype_` ， 那么该函数最后的查询一直执行到结束， 并且它输出的每一行都被当作该结果集中的一个元素返回。

这个特性通常用于把函数放在 `FROM` 子句里调用。 此时该函数返回的每一行都成为查询可见的该表的一行。 比如，假设表 `foo` 的内容和上面相同，那么:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

将得到：

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

它也有可能返回输出参数定义的列的多行，像这样：

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

这里关键的一点是你必须写 `RETURNS SETOF record` 表明函数返回多行而不是一行。 如果只有一个输出参数，写参数类型而不是 `record` 。

它通过调用多次设置返回函数构建一个查询结果经常是有用的，为了每个参数调用 一个表或查询连续的行。这样做的最佳方法是使用 `LATERAL` 关键字， 在Section 7.2.1.5中描述的。 这里是一个例子，使用设置返回函数来枚举树结构元素：

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

这个例子不做任何事情，我们不能做一个简单的连接，但在更复杂的计算中，选择把 一些工作放入一个函数中是很方便的。

目前，返回集合的函数也可以在一个查询的选择列表里调用。对于该查询自己生成的每一行，都会调用这个返回集合的函数，并且对于该函数的结果集中的每个元素都会生成一个输出行。不过，这个功能已经废弃了，在将来的版本中可能会被删除。下面就是一个在选择列表中使用返回集合的函数的例子：

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

请注意，在最后的 `SELECT` 里没有出现 `Child2` , `Child3` 等行。 这是因为 `listchildren` 为这些参数返回一个空集合， 因此不生成任何结果行。 当使用 `LATERAL` 语法时，这同从内部链接到函数结果行为是一样的。

> **Note:** 如果函数的最后命令是 `INSERT` , `UPDATE` , 或者带有 `RETURNING` 的 `DELETE` ，则命令将总是执行完成，即使 函数不被声明为 `SETOF` 或者调用查询不抓取所有结果行。 任何通过 `RETURNING` 子句产生的额外行静静地被删除，但是 仍然产生命令表修改（都是从函数返回前完成）。
>
> **Note:** 在选择列表中使用设置返回函数而不是 `FROM` 子句的关键问题是将一个以上的设置返回函数放在同一个选择列表中是不明智的。如果你将输出行数等同于 通过每个设置返回函数产生的行数的最小公倍数，你实际得到了什么。 当调用多个设置返回函数并且往往替代使用的时候， `LATERAL` 语法很少产生令人惊讶的结果。

35.4.9. 返回 `TABLE` 的SQL函数

还有另一种方式来声明返回集合的函数， 它是利用语法 `RETURNS TABLE(``_columns_)`。 这相当于使用一个或多个 `OUT` 参数加上 标记函数作为返回 `SETOF record` （或者 `SETOF` 一个输出参数的类型，视 情况而定）。这个符号是在最近的SQL标准版本中规定的，因此可能比使用 `SETOF` 更便捷。

比如，前面的和与乘积的例子可以这样做：

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

不允许使用明确的带有 `RETURNS TABLE` 标记的 `OUT` 或者 `INOUT` 参数— 你必须将所有输出参数放在 `TABLE` 列表中。

35.4.10. 多态SQL函数

SQL函数可以声明为接受并返回多态类型 `anyelement`, `anyarray`, `anynonarray`, `anyenum` 和 `anyrange` 。 参阅Section 35.2.5获取有关多态函数的更多细节。 下面是一个多态的函数 `make_array`， 它从两个任意数据类型元素中建立一个数组:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

请注意使用了类型转换 `'a'::text` 声明参数是 `text` 类型。 如果参数只是一个字符串文本，这是必须的， 否则它就会被当作 `unknown` 类型。 因为 `unknown` 不是一种有效的类型，所以如果没有类型转换， 就会看到类似下面这样的错误信息：

```
<samp class="literal">ERROR:  could not determine polymorphic type because input has type
```

允许含有多态参数的函数返回一个固定类型， 但是反过来不行。比如：

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic argum
```

多态性也可以用于那些含有输出参数的函数。比如：

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

多态性也可以使用可变参数函数。比如：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

35.4.11. 带有排序规则的SQL函数

当一个SQL函数具有一个或多个collatable数据类型的参数， 排序规则认同每个函数调用依赖于分配给实际参数的排序规则，正如Section 22.2描述的。如果 一个排序规则成功地被识别（即不存在参数之间的隐式排序规则的冲突） 然后所有的collatable参数作为 含蓄的排序规则对待。这会影响函数内排序规则区分操作行为。 例如，使用上文描述的 `anyleast` ，结果为

```
SELECT anyleast('abc'::text, 'ABC');
```

将依赖于数据库的缺省排序规则。在 `C` 中结果将是 `ABC` ， 但是在许多其他区域中它将是 `abc` 。使用的排序规则通过添加 `COLLATE` 子句强制给任何参数，比如

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

另外，如果你希望函数操作特定的排序规则不管称为什么， 作为需要插入 `COLLATE` 子句到函数定义中， `anyleast` 的版本可能总是使用 `en_US` 区域来比较字符串：

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

但是请注意如果适用于非-collatable数据类型，则将抛出一个错误。

如果在实际参数之间没有识别通用排序规则， 那么一个SQL函数将其参数作为数据类型 的默认排序规则（通常是数据库的默认排序规则，但不同于域类型参数）。

collatable参数操作可以被认为是多态的有限形式，只适用于文本数据类型。

35.5. 函数重载

多个函数可以定义成同样的SQL名字，只要它们接受的参数不同。 换句话说，函数名可以重载。在执行一个查询的时候， 服务器会从提供的参数类型和个数上判断应该调用哪个函数。重载也可以用于模拟数目不定(有上限)的参数。

在创建一族重载函数的时候，我们应该小心避免歧义。比如，对于下面的函数:

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

如果给出一些简单的输入，比如 `test(1, 1.5)` ， 系统要想判断应该调用哪个函数可不是一件很容易的事情。 目前实现的解析规则在Chapter 10里描述， 但是设计一个隐含依赖这些行为的系统是不明智的。

一个接受单个复合类型参数的函数通常不应该和该类型或该类型的任何属性(字段)同名。 `attribute(table)` 被认为等效于 `table.attribute` 。 在这种情况下，一个复合类型上的函数会和一个复合类型的属性有歧义(总是使用属性)。 我们可以通过使用模式来修饰函数名(也就是 `schema.func(table)`)以绕开这个限制， 但最好还是通过使用无冲突的名字来避免这个问题。

另一个可能的冲突是可变的和非可变参数之间的函数关系。 例如，它可以创建 `foo(numeric)` 和 `foo(VARIADIC numeric[])` 。在这种情况下，目前还不清楚哪一个 应匹配提供一个数字参数的调用，如 `foo(10.1)` 。规则是函数早期出现在使用的搜索路径中， 或者如果两个函数都在同样的模式下，非-可变参数优先。

在重载C语言函数的时候，还有一个额外的限制： 重载族的每个函数的C名字必须和所有其它函数的C名字不同， 其它函数包括内部的和动态加载的。如果违反这条规则， 那么行为是不可移植的。你可能会得到一个运行时的链接错误， 或者是其中一个函数被调用(通常是内部的那个)。 `CREATE FUNCTION` 命令可选的 `AS` 子句 把SQL函数名和C源代码中的函数名分离开。比如:

```
CREATE FUNCTION test(int) RETURNS int
    AS '_filename_', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS '_filename_', 'test_2arg'
    LANGUAGE C;
```

这里的C函数名代表许多可能的惯例之一。

35.6. 函数易失性范畴

每个函数都有一个易失性级别 `VOLATILE`， `STABLE` 或者 `IMMUTABLE`。 如果CREATE FUNCTION命令没有明确声明范畴的话， `VOLATILE` 就是缺省。易失性范畴是给优化器的一个关于函数行为的承诺：

- `VOLATILE` 可以做任何事情，包括修改数据库。 它可以在使用同样参数调用时返回不同的结果。优化器对这样的函数不做任何假设。 一个使用易失函数的查询在需要数据值的时候每次都重新计算函数的值。
- `STABLE` 函数不会修改数据库， 并且保证在同一个查询的环境里给出相同参数的情况下，会给出相同的结果。 这个范畴允许优化器在一个查询里把多个函数调用优化成一个。 特别是在索引扫描的条件表达式里面包含这样的函数是安全的。 因为索引扫描只计算一次比较值，而不是每行一次。 在索引扫描条件里使用一个 `VOLATILE` 函数是非法的。
- `IMMUTABLE` 函数不会修改数据库， 并且保证在任何情况下同样的参数永远返回同样的结果。 这个范畴允许优化器在查询调用函数的时候预先把函数计算成一个常量参数。 比如，类似 `SELECT ... WHERE x = 2 + 2` 的查询可以简化成 `SELECT ... WHERE x = 4`，因为在加法操作符下层的函数是标记为 `IMMUTABLE` 的。

为了最佳的优化结果，应该尽可能使用最严格的易失性范畴标记你的函数。

任何有副作用的函数都必须标记为 `VOLATILE`， 这样对它的调用就不会被优化。即使一个函数没有副作用， 但它的数值可能在一个查询里改变，那么也必须标记为 `VOLATILE` ；例如 `random()`， `currval()`， `timeofday()` 函数。

另一个重要例子是 `current_timestamp` 函数簇描述为 `STABLE`， 因为他们的值在一个事务中没有改变。

在那些简单的规划后马上执行的交互查询上， `STABLE` 和 `IMMUTABLE` 没有什么区别： 函数是在规划开始时执行还是在查询开始时执行的差别并不大。 但是如果规划被保存并且后来被重用，那差别可就大了。 如果把一个函数标记为 `IMMUTABLE` 而它实际上又不是， 那么就会导致在随后使用其规划的时候用上一个不完整的数值。 如果在使用预先准备好语句或者使用一种缓冲规划的函数语言(比如PL/pgSQL)， 那么后果可能很严重。

为了编写SQL或在任何标准的程序语言的函数，还有一个通过波动性范畴决定的重要属性，即由SQL命令决定的任何数据变化能见度正在调用函数。 一个 `VOLATILE` 函数将看到这样的变化， `STABLE` 或者 `IMMUTABLE` 函数不这样。这种行为 使用MVCC快照行为实现（参阅Chapter 13） ： `STABLE` 和 `IMMUTABLE` 函数使用快照 确立为调用查询的开始， 而 `VOLATILE` 函数每个查询执行开始时获得一个新的快照。

> **Note:** 用C写的函数管理快照然而是他们想要的，但是使C函数这样进行工作往往是一个好主意。

因为快照行为， 一个只包含 `SELECT` 命令的函数可以安全地标记为 `STABLE` ， 即使它所选择的表可能会被其它并发查询修改也一样。 PostgreSQL将会在执行 `STABLE` 函数时为调用它的查询建立快照， 因此它在该查询的生存期内都会看到一致的数据库视图。

同样的快照行为也用于 `IMMUTABLE` 函数里面的 `SELECT` 命令。 通常，在一个 `IMMUTABLE` 函数里选择一个数据库的表是不明智的， 因为如果表的内容改变，那么这种不变性就将改变。不过， PostgreSQL并不禁止你这样做。

一个常见的错误是把一个函数标记为 `IMMUTABLE` ， 而实际上这个函数的结果依赖某个配置参数。比如， 一个操作时间戳的函数可能有依赖于TimeZone设置的结果。 为了安全考虑，这样的函数应该标记为 `STABLE` 。

> **Note:** 在PostgreSQL之前，要求 `STABLE` 和 `IMMUTABLE` 函数不能修改数据库这个约束并未由系统强制。版本8.0 通过要求这类函数不能包含 `SELECT` 之外的 SQL 命令来强制这个约束。 不过，这么做并不是完全防弹的升级， 因为这样的函数仍然可以调用那些可能修改数据库的 `VOLATILE` 函数。 如果你这么做的话将会发现 `STABLE` 或者 `IMMUTABLE` 并不会觉察到被它调用的函数对数据库所做的修改。

35.7. 过程语言函数

PostgreSQL允许用 SQL 和 C 之外的语言书写用户自定义的函数。 这样的语言通常被称为过程语言 (PLs)。 过程语言函数不是内建于PostgreSQL里的。 它们是通过可加载模块提供的。参阅Chapter 39和随后的章节获取更多信息。

35.8. 内部函数

内部函数都是用C写的函数，它们已经通过静态链接的方式嵌入 PostgreSQL服务器进程中了。 函数定义的"函数体"确定了函数的C语言名称， 它不必与给 SQL 使用的名称相同。出于向下兼容考虑， 一个空的函数体也可以被接受， 这意味着 C 函数名与 SQL 函数名相同。

通常，所有在服务器里出现的内部函数都在数据库初始化时定义(参阅Section 17.2)。 但是用户可以用 `CREATE FUNCTION` 为内部函数创建额外的别名。 内部函数在 `CREATE FUNCTION` 命令里是带着 `internal` 语言名声明的。 比如，要给 `sqrt` 函数创建一个别名：

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

（大多数内部函数都应该声明为"strict"）。

> **Note:** 并非所有"预定义"的函数都像上面那样是"内部的"。 有些预定义的函数是用 SQL 写的。

35.9. C-语言函数

用户定义的函数可以用 C 写(或者是与C兼容的语言，比如C++)。 这样的函数被编译进动态加载对象(共享库)并且由服务器根据需要加载。 动态加载的特性是"C 语言函数"和"内部函数"之间的区别—不过， 实际的编码习惯在两者之间实际上是一样的。因此， 标准的内部函数库为写用户定义C函数提供了大量最好的样例。

目前对 C 函数有两种调用约定。新的"版本-1"的调用约定是通过为该函数书写一个 `PG_FUNCTION_INFO_V1()` 宏来标识的，像下面演示的那样。缺少这个宏表示一个 老风格的("版本-0")函数。两种风格里在 `CREATE FUNCTION` 里声明的都是 `C` 。 现在老风格的函数已经废弃了，主要是因为移植性原因和缺乏功能， 不过出于兼容性原因，系统仍然支持它。

35.9.1. 动态加载

当用户定义的函数第一次被服务器会话调用时， 动态加载器才把可加载对象文件里的函数目标码加载进内存。 因此，用于用户定义 C 函数的 `CREATE FUNCTION` 必须为函数声明两个信息： 可加载对象文件名、在目标文件里调用的 C 函数名(连接符号)。 如果没有明确声明 C 函数名，那么就假设它与 SQL 函数名相同。

基于在 `CREATE FUNCTION` 命令中给出的名字， 下面的算法用于定位共享对象文件：

1. 如果名字是一个绝对路径，则加载给出的文件。
2. 如果名字以字符串 `$libdir` 开头， 那么该部分将被PostgreSQL库目录名代替， 该目录是在编译时确定的。
3. 如果名字不包含目录部分， 那么在配置参数dynamic_library_path. 声明的路径里查找。
4. 如果没有在路径里找到该文件，或者它包含一个非绝对目录部分， 那么动态加载器就会试图直接拿这个名字来加载， 这样几乎可以肯定是要失败的(依靠当前工作目录是不可靠的)。

如果这个顺序不管用，那么就给这个名字加上平台相关的共享库文件扩展名(通常是 `.so`)，然后再重新按照上面的过程找一遍。如果还是失败，那么加载失败。

建议使用相对于 `$libdir` 的目录或者通过动态库路径定位共享库。 这样，如果新版本安装在一个不同的位置，那么就可以简化版本升级。 `$libdir` 的实际目录位置可以用 `pg_config --pkglibdir` 命令找到。

运行PostgreSQL服务器的用户ID 必须可以遍历路径到达想加载的文件。 一个常见的错误就是把该文件或者一个高层目 录的权限设置为postgres用户不可读和/或不能执行。

在任何情况下，在 `CREATE FUNCTION` 命令里给出的文件 名是在系统表里按照文本记录的， 因此，如果需要再次加载，那么会再次运行这个过程。

> **Note:** PostgreSQL不会自动编译 C 函数； 在使用 `CREATE FUNCTION` 命令之前你必须编译它。 参阅Section 35.9.6获取更多信息。

为了确保不会错误加载共享库文件，从PostgreSQL 开始将检查那个文件的"magic block"，这允许服务器以检查明显的不兼容性。 比如不同版本PostgreSQL的编译代码。 magic block需要被作为PostgreSQL 8.2。 为了包含"magic block"， 请在包含了 `fmgr.h` 头文件之后， 将下面的内容写进一个(也只能是一个)模块的源代码文件中：

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

如果不打算兼容8.2 PostgreSQL之前的版本， `#ifdef` 测试也可以省略。

动态加载对象文件在首次使用之后将一直滞留在内存中。 在同一个会话中的下一次调用将只需查找符号表的很小开销。 如果你想强制重新加载(比如重新编译之后)， 可以重新开始一个新的会话。

动态加载文件也可以包含初始化函数和结束函数。 如果包含一个名为 `_PG_init` 的函数，那么该函数将在该文件被加载后立即执行， 该函数不能接受任何参数并且必须返回 void 。 如果包含一个名为 `_PG_fini` 的函数，那么该函数将在该文件即将被卸载前执行， 同样，该函数不能接受任何参数并且必须返回 void 。 需要注意的是 `_PG_fini` 仅在该文件即将被卸载前执行而不是在会话结束的时候执行。 目前，卸载被禁止并且将不会发生，但是这可能在将来改变。

35.9.2. 基本类型的C语言函数

要知道如何写C语言函数，就必须知道PostgreSQL在 内部如何表现基本数据类型以及如何传入及传出函数。 PostgreSQL内部把基本类型当作"一块内存"看待。 定义在某种类型上的用户定义函数实际上定义了 PostgreSQL对该数据类型可能的操作。也就是说， PostgreSQL只是从磁盘读取和存储该数据 类型并使用你定义的函数来输入、处理、输出数据。

基本类型可以有下面三种内部形态(格式)之一：

- 传递数值，定长
- 传递引用，定长
- 传递引用，变长

传递数值的类型长度只能是1, 2, 4字节。如果 `sizeof(Datum)` 在你的机器上是8的话，那么还有8字节。你要仔细定义你的类型， 确保它们在任何体系平台上都是相同尺寸(字节)。例如， `long` 是一个危险的类型， 因为在一些机器上它是4字节而在另外一些机器上是8字节，而 `int` 在大多数Unix机器上都是4字节的。 在一个Unix机器上的 `int4` 合理实现可能是：

```
/* 4-byte integer, passed by value */
typedef int int4;
```

实际PostgreSQL C代码调用此 `int32` 类型， 因为它是C中的惯例， `int``_XX_` 意味着 _XX_ *bits*。 注意因此C类型 `int8` 的大小是1字节。SQL类型 `int8` 被称为C中 `int64` 。参见 Table 35-1。

另外，任何尺寸的定长类型都可以是传递引用型。例如， 下面是一个PostgreSQL类型的实现：

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

只能使用指向这些类型的指针在PostgreSQL函数里传入和传出数据。 要返回这样类型的值，用 `palloc` 分配正确数量的内存，填充这些内存， 然后返回一个指向它的指针。如果只是想返回和输入参数类型与数值都相同的数值， 可以忽略额外的 `palloc` ，只要返回指向输入数值的指针就行。

最后，所有变长类型同样也只能通过引用来传递。 所有变长类型必须以一个4字节的长度域开始，通过 `SET_VARSIZE` 设置， 没有直接设置这个字段！ 所有存储在该类型中的数据必须放在紧接着长度域的存储空间里。 长度域是结构的全长，也就是说，包括长度域本身的长度。

另外一个重要的点是避免数据类型值中留下未初始化的位；比如，请注意任何对齐填充字节溢出的零可能出现在结构体中。没有这些，你的数据类型的逻辑上等价常量可能被规划器看做是 不平等的，导致低效（虽然是不正确的）规划。

Warning

绝对不要修改一个引用传递的输入值，否则很可能破坏磁盘上的数据。 因为指针很可能直接指向一个磁盘缓冲区。 这条规则的唯一例外在Section 35.10里。

比如，我们可以用下面的方法定义一个 `text` 类型:

```
typedef struct {
    int32 length;
    char data[1];
} text;
```

显然，上面声明的数据域长度不足以存储任何可能的字符串。 因为在C中不可能声明变长结构，所以我们倚赖这样的知识：C编译器不会对数组下标进行范围检查。只需要分配足够的空间， 然后把数组当做已经声明为合适长度的变量访问。这是一个常用的技巧， 你可以在许多C教科书中读到。

当处理变长类型时，必须仔细分配正确的内存数量并正确设置长度域。 例如，如果想在一个 `text` 结构里存储40字节， 我们可能会使用像下面的代码片段：

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` 等价于 `sizeof(int32)`， 但是我们认为用宏 `VARHDRSZ` 表示附加尺寸是用于变长类型的更好风格。 同时，该长度字段必须使用 `SET_VARSIZE` 宏设置， 而不是简单的赋值。

Table 35-1列出了书写 使用PostgreSQL内置类型的 C 函数里需要知道的 SQL 类型与 C 类型的对应关系。"定义在" 列给出了需要包含以获取该类型定义的头文件。 实际定义可能在列表文件中包含的不同文件中。我们建议用户坚持定义的接口。 注意，你应该总是首先包括 `postgres.h`， 因为它声明了许多你需要的东西。

Table 35-1. 与内建SQL类型等效的C类型

SQL Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h (可能编译器内置)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(编译器内置)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

既然我们已经讨论了基本类型所有可能的结构， 我们便可以用实际的函数举一些例子。

35.9.3. 版本-0调用约定

先提供现在已经不提倡了的"老风格"—因为比较容易迈出第一步。 在版本-0方法中，此风格 C 函数的参数和结果用普通 C 风格声明， 但是要小心使用上面显示的 SQL 数据类型的 C 表现形式。

下面是一些例子：

```
#include "postgres.h"
#include <string.h>
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* 传递数值 */
int
add_one(int arg)
{
    return arg + 1;
}

/* 传递引用，定长 */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* 传递引用，变长*/

text *
copytext(text *t)
{
 /*
     * VARSIZE是结构以字节计的总长度。
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));

/*
     * VARDATA是结构中一个指向数据区的指针。
     */

    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
```

```
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}
```

假设上面的代码放在 `funcs.c` 文件中并且编译成了共享目标， 我们可以用下面的命令为PostgreSQL定义这些函数:

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS '_DIRECTORY_/funcs', 'add_one'
     LANGUAGE C STRICT;
--注意：重载了名字为"add_one"的 SQL 函数

CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS '_DIRECTORY_/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS '_DIRECTORY_/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS '_DIRECTORY_/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS '_DIRECTORY_/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

这里的 `_DIRECTORY_` 代表共享库文件的目录， 比如包含本节示例代码的PostgreSQL教程目录。 更好的风格应该是将 `_DIRECTORY_` 加到搜索路径之后， 在 `AS` 子句里只使用 `'funcs'` ， 不管怎样， 我们都可以省略和系统相关的共享库扩展， 通常是 `.so` 或者 `.sl` 。

请注意我们把函数声明为"strict"(严格)，意思是说如果任何输入值为NULL， 那么系统应该自动假设一个NULL的结果。这样处理可以让我们避免在函数代码里面检查 NULL输入。如果不这样处理，我们就得明确检查NULL， 比如为每个传递引用的参数检查空指针。对于传值类型的参数，我们甚至没有办法检查！

尽管这种老调用风格用起来简单，但它却不太容易移植； 在一些系统上，用这种方法传递比 `int` 小的数据类型就会碰到困难。 而且，我们没有很好的返回NULL结果的办法，也没有除了把函数严格化以外的处理 NULL参数的方法。版本-1约定，下面要讲的新方法则解决了这些问题。

35.9.4. 版本1调用约定

版本-1调用约定使用宏消除大多数传递参数和结果的复杂性。版本-1风格函数的C定义总是下面这样:

```
Datum funcname(PG_FUNCTION_ARGS)
```

另外，宏调用：

```
PG_FUNCTION_INFO_V1(funcname);
```

也必须出现在同一个源文件里(通常就可以写在函数自身前面)。 对那些 `internal` 语言函数而言，不需要调用这个宏， 因为PostgreSQL目前假设内部函数都是版本-1。不过，对于动态加载的函数， 它是必须的。

在版本-1 函数里，每个实际参数都是用一个对应该参数的数据类型的 `PG_GETARG_` *xxx* `()` 宏抓取的， 用返回类型的 `PG_RETURN_` *xxx* `()` 宏返回结果。 `PG_GETARG_` *xxx* `()` 接受要抓取的函数参数的编号 (从 0 开始)作为其参数。 `PG_RETURN_` *xxx* `()` 接受实际要返回的数值为自身的参数。

下面是和上面一样的函数，但是使用版本-1风格编写的：

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/*传递数值*/

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/*传递引用，定长*/

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
 /*用于FLOAT8的宏，隐藏其传递引用的本质。*/
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
 /* 这里，我们没有隐藏Point的传递引用的本质*/
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));
```

```
    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/*传递引用，变长*/

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);

 /*
     * VARSIZE是结构以字节计的总长度。
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));

 /*
     * VARDATA是结构中指向数据区的一个指针。
     */

memcpy((void *) VARDATA(new_t), /* 目的*/
           (void *) VARDATA(t),      /* 源 */
           VARSIZE(t) - VARHDRSZ);  /* 多少字节 */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

用到的 `CREATE FUNCTION` 命令和版本-0等效命令一样。

猛一看，版本-1的编码好像只是无目的地蒙人。但是它的确给我们许多改进， 因为宏可以隐藏许多不必要的细节。一个例子在 `add_one_float8` 的编码里， 这里我们不再需要不停叮嘱自己 `float8` 是传递引用类型。 另外一个例子是用于变长类型的宏 `GETARG` 隐藏了抓取 "非常规"(压缩的或者超长的)值需要做的处理。

版本-1的函数另一个巨大的改进是对NULL输入和结果的处理。 宏 `PG_ARGISNULL(` *n* `)` 允许一个函数测试每个输入是否为NULL， 当然，这只是对那些没有声明为"strict"的函数有必要。 因为如果有 `PG_GETARG_` *xxx* `()` 宏， 输入参数是从零开始计算的。 请注意我们不应该执行 `PG_GETARG_` *xxx* `()` ， 除非有人声明了参数不是NULL。 要返回一个NULL结果，可以执行一个 `PG_RETURN_NULL()` ， 这样对严格的和不严格的函数都有效。

在新风格的接口中提供的其它选项是 `PG_GETARG_` *xxx* `()` 宏的两个变种。 第一个变体 `PG_GETARG_` *xxx* `_COPY()` 保证返回一个指定参数的副本，该副本是可以安全地写入的。普通的宏有时候会返回一个指向物理存储在表中的某值的指针， 因此我们不能写入该指针。用 `PG_GETARG_` *xxx* `_COPY()` 宏保证获取一个可写的结果。 第二个变体由 `PG_GETARG_` *xxx* `_SLICE()` 宏组成， 它接受三个参数。第一个是参数的个数(与上同)。第二个和第三个是要返回的偏移量和数据段的长度。 偏移是从零开始计算的，一个负数的长度则要求返回该值的剩余长度的数据。 这些过程提供了访问大数据值的中一部分的更有效方法，特别是数据的存储类型是"external"的时候。 一个字段的存储类型可以用 `ALTER TABLE _tablename_` ALTER COLUMN `_colname_` SET STORAGE `_storagetype_` 指定。 `_storagetype_` 是 `plain` ， `external` ， `extended` ， 或者 `main` 之一。

版本-1 的函数调用风格也令我们可能返回一"套"结果(Section 35.9.9) 并且实现触发器函数 (Chapter 36)和过程语言调用处理器 (Chapter 51)。 版本-1的代码也更容易移植，因为它没有违反C标准对函数调用协议的限制。 更多的细节请参阅源程序中的 `src/backend/utils/fmgr/README` 文件。

35.9.5. 书写代码

在转到更深的话题之前，先要讨论一些PostgreSQL C语言函数的编码规则。 虽然可以用C以外的其它语言书写用于 PostgreSQL的共享函数， 但通常都很麻烦（当它可能的时候）， 因为其他语言， 比如C++, FORTRAN或者Pascal并不遵循C的调用习惯。 也就是说，其它语言在函数之间的传递参数和返回值的方式不一样。 因此假设你的C-编程语言函数是用C写的。

书写和编译C函数的基本规则如下：

- 使用 `pg_config --includedir-server` 找出PostgreSQL服务器的头文件安装在你的系统上的 （或者你的用户正在运行）的位置。
- 把你的代码编译成可以动态装入PostgreSQL 的库文件总是需要一些特殊的标记。参阅Section 35.9.6获取如何在你的平台上做这件事的详细说明。
- 按照Section 35.9.1的指示为你的共享库定义一个"magic block"。
- 当分配内存时，用PostgreSQL的 `palloc` 和 `pfree` 函数取代相应的C库函数 `malloc` 和 `free` 。 用 `palloc` 分配的内存在每个事务结束时会自动释放，避免了内存泄露。
- 使用 `memset` （或者在第一个位置分配 `palloc0` ）的你的结构字节总是零。 即使你给结构分配每个字段，可能有对齐填充（结构中含有孔）包含垃圾值。如果没有这一点， 很难支持散列索引和哈希连接，你必须只挑出你的数据结构中重要的位来计算一个散列。 规划器有时也依赖于通过位平等比较常数，所以如果逻辑等效值不是按位平等， 则你可能得到不良的规划结果。

- 大多数的PostgreSQL内部类型定义在 `postgres.h` 中， 而函数管理器接口(`PG_FUNCTION_ARGS` 等等)都在 `fmgr.h` 中， 所以你至少应该包括这两个文件。出于移植性原因， 最好先包括 `postgres.h` 再包含其它系统或者用户头文件。 包含 `postgres.h` 将自动包含 `elog.h` 和 `palloc.h` 。
- 在目标文件里定义的符号一定不能相互冲突， 也不能和定义在PostgreSQL服务器可执行代码中的符号名字冲突。 如果你看到了与此相关的错误信息，那么必须重命名你的函数或者变量。

35.9.6. 编译和链接动态加载的函数

在能够使用由 C 写的PostgreSQL扩展函数之前， 必须用一种特殊的方法编译和链接它们，这样才能生成可以被服务器动态加载的文件。 准确地说是需要创建一个共享库。

如果需要更多信息，那么你应该阅读操作系统的文档，特别是 C 编译器(`cc`) 和连接器(`ld`) 的文档。另外，PostgreSQL 源代码里包含几个可以运行的例子，它们在 `contrib` 目录里。不过， 如果你依赖这些例子，那么你的模块将依赖于PostgreSQL源代码的可用性。

创建共享库和链接可执行文件类似：首先把源代码编译成目标文件，然后把目标文件链接起来。 目标文件需要创建成位置无关码(PIC)， 也就是在可执行程序加载它们的时候， 它们可以被放在可执行程序内存里的任何地方(用于可执行文件的目标文件通常不是用这个方式编译的)， 链接动态库的命令包含特殊标志，与链接可执行文件的命令是有区别的(至少理论上如此，不过现实未必)。

在下面的例子里，假设你的源程序代码在 `foo.c` 文件里，并且我们要创建 `foo.so` 的共享库。中介的对象文件将叫做 `foo.o` (除非另外注明)。虽然一个共享库可以包含多个对象文件，但是在这里只用一个。

FreeBSD

创建PIC的编译器标志是 `-fpic` 。创建共享库的链接器标志是 `-shared` 。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

上面方法适用于 3.0 版本的FreeBSD。

HP-UX

创建PIC的编译器标志是 `+z` 。如果使用GCC 则是 `-fpic` 。创建共享库的链接器标志是 `-b` 。因此：

```
cc +z -c foo.c
```

或：

```
gcc -fpic -c foo.c
```

然后：

```
ld -b -o foo.sl foo.o
```

HP-UX使用 `.sl` 作为共享库扩展名，和其它大部分系统不同。

IRIX

PIC是缺省，不需要使用特殊的编译器选项。创建共享库的链接器标志是 `-shared` 。

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

创建PIC的编译器标志是 `-fpic` 。在某些平台上如果 `-fpic` 不工作则必须使用 `-fPIC` 。 参考 GCC 手册获取更多信息。创建共享库的编译器标志是 `-shared` 。一个完整的例子看起来像：

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

Mac OS X

这里是一个例子。假设开发工具已经安装好了。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

创建PIC的编译器标志是 `-fpic` 。对于ELF系统， 带 `-shared` 标志的编译命令用于链接共享库。在老的非 ELF 系统里， 则使用 `ld -Bshareable` 。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

创建PIC的编译器标志是 `-fpic` 。而 `ld -Bshareable` 用于链接共享库。

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

用 Sun 编译器时创建PIC的编译器标志是 `-KPIC` ；用GCC编译器时创建PIC的编译器标志是 `-fpic` 。 链接共享库时两个编译器都可以用 `-G` ， 此外GCC还可以用 `-shared` 。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

或

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC是缺省，不需要使用特殊的编译器选项。带特殊选项的 `ld` 用于链接：

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

用 GCC 代替系统编译器时的过程是一样的；不需要特殊的选项。

UnixWare

用 SCO 编译器时创建PIC的编译器标志是 `-K PIC` ； 用GCC编译器时创建PIC的编译器标志是 `-fpic` 。链接共享库时 SCO 编译器用 `-G` 而GCC 使用 `-shared` 。

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

或

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

> **Tip:** 如果你觉得这些步骤实在太复杂，那么你应该考虑使用GNU Libtool，它把平台的差异隐藏在了一个统一的接口里。

生成的共享库文件然后就可以加载到PostgreSQL里面去了。 在给 `CREATE FUNCTION` 命令声明文件名的时候， 必须声明共享库文件的名字而不是中间目标文件的名字。 请注意你可以在 `CREATE FUNCTION` 命令上忽略系统标准的共享库扩展名(通常是 `.so` 或 `.sl`)，并且出于最佳的兼容性考虑也应该忽略。

回头看看Section 35.9.1获取有关服务器预期在哪里找到共享库的信息。

35.9.7. 复合类型参数

复合类型不像 C 结构那样有固定的布局。复合类型的实例可能包含空(NULL)字段。另外， 一个属于继承层次一部分的复合类型可能和同一继承范畴的其它成员有不同的域/字段。 因此，PostgreSQL提供一个过程接口用于从C中访问复合类型。

假设为下面查询写一个函数:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

使用调用约定版本0，可以这样定义 `c_overpaid` ：

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

bool
c_overpaid(HeapTupleHeader t, /* the current row of emp */
           int32 limit)
{
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return false;
    return salary > limit;
}
```

如果用版本-1则会写成下面这样:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
<!--
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */
-->

    /*另外，可能更希望将PG_RETURN_NULL()用在null薪水上*/

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

`GetAttributeByName` 是PostgreSQL系统函数， 用来返回当前记录的字段。它有三个参数：类型为 `HeapTupleHeader` 的传入函数的参数、你想要的字段名称、 一个确定字段是否为 NULL 的返回参数。 `GetAttributeByName` 函数返回一个 `Datum` 值，你可以用对应的 `DatumGet`*XXX*`()` 宏把它转换成合适的数据类型。 请注意，如果设置了NULL标志，那么返回值是无意义的，在准备对结果做任何处理之前， 总是要先检查NULL标志。

还有一个 `GetAttributeByNum` 用字段编号而不是字段名选取目标字段。

下面的命令在SQL里声明 `c_overpaid` 函数：

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS '_DIRECTORY_/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

请注意使用 `STRICT` 后就不需要检查输入参数是否有NULL。

35.9.8. 返回行(复合类型)

要从一个C语言函数里返回一个行或复合类型的数值， 可以使用一个特殊的API，它提供了许多宏和函数来消除大多数制作复合数据类型的复杂性。 要使用该API，源代码必须包含：

```
#include "funcapi.h"
```

制作一个复合类型数据值(也就是一个"行")有两种方法： 你可以从一个 Datum 值数组里制作，也可以从一个可以传递给该行的字段类型的输入转换函数的 C 字符串数组里制作。不管是哪种方式，你首先都需要为行结构获取或者制作一个 `TupleDesc` 描述符。在使用 Datums 的时候，你给 `BlessTupleDesc` 传递这个 `TupleDesc` 然后为每行调用 `heap_form_tuple` 。在使用C字符串的时候， 你给 `TupleDescGetAttInMetadata` 传递 `TupleDesc` ，然后为每行调用 `BuildTupleFromCStrings` 。 如果是返回一个行集合的场合，所有设置步骤都可以在第一次调用该函数的时候一次性完成。

有几个便利函数可以用于设置所需要的 `TupleDesc` 。 在大多数返回复合类型给调用者的函数里建议的做法是这样的：

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

把传递给调用函数自己的 `fcinfo` 传递给它(要求使用版本-1 的调用习惯)。 `resultTypeId` 可以声明为 `NULL` 或者 接收函数的结果类型OID的局部变量地址(指针)。 `resultTupleDesc` 应该是一个局部的 `TupleDesc` 变量地址(指针)。 检查结果是否 `TYPEFUNC_COMPOSITE` ；如是， `resultTupleDesc` 就已经填充好需要的 `TupleDesc` 了。 如果不是，你可以报告一个类似"返回记录的函数在一个不接受记录的环境中被调用"的错误。

> **Tip:** `get_call_result_type` 可以把一个多态的函数结果解析为实际类型； 因此它在返回多态的标量结果的函数里也很有用，而不仅仅是返回复合类型的函数里。 `resultTypeId` 输出主要用于那些返回多态的标量类型的函数。
>
> **Note:** `get_call_result_type` 有一个同胞弟兄 `get_expr_result_type` 可以用于给一个用表达式树表示的函数调用解析输出， 它可以用于视图从函数本身外边判断结果类型的场合。 还有一个 `get_func_result_type` 可以用在只能拿到函数OID的场合。 不过， 这些函数不能处理那些声明为返回 `record` 的函数， 并且 `get_func_result_type` 不能解析多态的类型， 因此你最好还是使用 `get_call_result_type` 。

旧的，现在已经废弃的获取 `TupleDesc` 的函数是:

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

它可以从一个命名的关系里为行类型获取一个 `TupleDesc` ， 还有:

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

可以基于类型 OID 获取一个 `TupleDesc` 。 它可以用于给一个基本类型或者一个复合类型获取 `TupleDesc` 。 不过它不能处理返回 `record` 的函数，并且不能解析多态的类型。

一旦你有了一个 `TupleDesc` ，那么调用:

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

如果你想使用Datum，或者：

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

如果你想使用C字符串。如果你在写一个返回集合的函数， 那么你可以把这些函数的结果保存在 FuncCallContext 结构里 (分别使用 tuple_desc 或者 attinmeta 字段)。

在使用Datum的时候，使用：

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

制作一个 HeapTuple ，它把数据以Datum的形式交给用户。

当使用C字符串时，使用：

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

制作一个 HeapTuple ，以C字符串的形式给出用户数据。 values 是一个 C 字符串的数组，返回行的每个字段对应其中一个。 每个 C 字符串都应该是字段数据类型的输入函数预期的形式。 为了从其中一个字段中返回一个NULL， values 数组中对应的指针应该设置为 NULL 。 这个函数将会需要为你返回的每个行调用一次。

一旦你制作了一个从你的函数中返回的行，那么该行必须转换成一个 Datum 。使用:

```
HeapTupleGetDatum(HeapTuple tuple)
```

把一个 HeapTuple 转换为一个有效的 Datum 。 如果你想只返回一行，那么这个 Datum 可以用于直接返回， 或者是它可以用作在一个返回集合的函数里的当前返回值。

例子在下面给出。

35.9.9. 返回集合

还有一个特殊的API用于提供从C语言函数中返回集合(多行)。 一个返回集合的函数必须遵循版本-1的调用方式。 同样，源代码必须包含 funcapi.h ，就像上面说的那样。

一个返回集合的函数(SRF)通常为它返回的每个项都调用一次。 因此SRF必须保存足够的状态用于记住它正在做的事情以及在每次调用的时候返回下一个项。 表函数 API 提供了 FuncCallContext 结构用于帮助控制这个过程。 fcinfo->flinfo->fn_extra 用于保存一个跨越多次调用的指向 FuncCallContext 的指针。

```
typedef struct
{

 /*
     * 前面已经被调用的次数
     * 初始的时候，call_cntr 被 SRF_FIRSTCALL_INIT() 置为 0，
 *并且每次你调用 SRF_RETURN_NEXT() 的时候都递增
     */
    uint32 call_cntr;

/*
     * 可选的最大调用数量
     * 这里的 max_calls 只是为了方便，设置它也是可选的。
     * 如果没有设置，你必须提供可选的方法来知道函数何时结束。
     */
    uint32 max_calls;

/*
     * 指向结果槽位的可选指针
     * 这个数据类型已经过时，只用于向下兼容。也就是那些使用已废弃的TupleDescGetSlot()的用户定义 SRF
     */
    TupleTableSlot *slot;

/*
     * 可选的指向用户提供的杂项环境信息的指针
     * user_fctx 用做一个指向你自己的结构的指针，包含任意提供给你的函数的调用间的环境信息
     */
    void *user_fctx;

 /*
     * 可选的指向包含属性类型输入元信息的结构数组的指针
     * attinmeta 用于在返回行的时候(也就是说返回复合数据类型)
     * 在只返回基本(也就是标量)数据类型的时候并不需要。
     * 只有在你准备用 BuildTupleFromCStrings() 创建返回行的时候才需要它。
     */

    AttInMetadata *attinmeta;

 /*
     * 用于必须在多次调用间存活的结构的内存环境
     * multi_call_memory_ctx 是由 SRF_FIRSTCALL_INIT() 为你设置的，并且由 SRF_RETURN_DONE() 月
     * 它是用于存放任何需要跨越多次调用 SRF 之间重复使用的内存。
     */
    MemoryContext multi_call_memory_ctx;

/*
     * 可选的指针，指向包含行描述的结构
     * tuple_desc 用于返回行(也就是说复合数据类型)并且只是在你想使用 heap_form_tuple() 而不是 Buil
     * 请注意这里存储的 TupleDesc 指针通常应该先用 BlessTupleDesc() 处理。
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

一个SRF使用自动操作 `FuncCallContext` 结构 (可以通过 `fn_extra` 找到)的若干个函数和宏。使用：

```
SRF_IS_FIRSTCALL()
```

来判断你的函数是第一次调用还是后继的调用。只有在第一次调用的时候，使用：

```
SRF_FIRSTCALL_INIT()
```

初始化 `FuncCallContext` 。在每次函数调用时(包括第一次)，使用：

```
SRF_PERCALL_SETUP()
```

为使用 `FuncCallContext` 做恰当的设置以及清理任何前面的轮回里面剩下的已返回的数据。

如果你的函数有数据要返回，使用:

```
SRF_RETURN_NEXT(funcctx, result)
```

返回给调用者(`result` 必须是个 `Datum` ，要么是单个值， 要么是像前面介绍的那样准备的行)。 最后，如果你的函数结束了数据返回，使用:

```
SRF_RETURN_DONE(funcctx)
```

清理并结束SRF。

在SRF被调用时的内存环境是一个临时环境， 在调用之间将会被清理掉。 这意味着你不需要 `pfree` 所有你 `palloc` 的东西；它会自动消失的。 不过，如果你想分配任何跨越调用存在的数据结构， 那你就需要把它们放在其它什么地方。 被 `multi_call_memory_ctx` 引用的环境适合用于保存那些需要直到 SRF结束前都存活的数据。在大多数情况下， 这意味着你在第一次调用设置的时候应该切换到 `multi_call_memory_ctx` 。

一个完整的伪代码例子看起来像下面这样：

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;

_更多的声明_

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
/* 这里放出现一次的设置代码: */
        _用户代码_
        _if 返回复合_
            _制作 TupleDesc 以及可能还有 AttInMetadata_
        _endif 返回复合_
        _用户定义代码_
        MemoryContextSwitchTo(oldcontext);
    }

/* 每次都执行的设置代码在这里出现: */
    _用户定义代码_
    funcctx = SRF_PERCALL_SETUP();
    _用户定义代码_
    /* 这里只是用来测试是否完成的一个方法: */

    if (funcctx->call_cntr < funcctx->max_calls)
    {

/* 这里想返回另外一个条目: */
        _用户代码_
        _获取结果_

        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {

/* 这里完成返回条目的工作了，只需要清理就OK了: */
        _用户代码_

        SRF_RETURN_DONE(funcctx);
    }
}
```

一个返回复合类型的完整SRF例子看起来像这样：

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

/* 只是在第一次调用函数的时候干的事情 */

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;

/*创建一个函数环境，用于在调用间保持住*/
```

```
        funcctx = SRF_FIRSTCALL_INIT();

/* 切换到适合多次函数调用的内存环境 */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

/* 要返回的行总数 */
        funcctx->max_calls = PG_GETARG_UINT32(0);

 /* 为了结果类型制作一个行描述 */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
                            "that cannot accept type record")));

 /*
         * 生成稍后从裸 C 字符串生成行的属性元数据
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

/* 每次函数调用都要做的事情 */

    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* 在还有需要发送的东西时继续处理 */

    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

/*
         * 准备一个数值数组用于版本的返回行
         * 它应该是一个C字符串数组，稍后可以被合适的类型输入函数处理。
         */

        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

/* 制作一个行 */
        tuple = BuildTupleFromCStrings(attinmeta, values);

/* 把行做成 datum  */
        result = HeapTupleGetDatum(tuple);

/* 清理(这些实际上并非必要) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }

 else    /* 在没有数据残留的时候干的事情 */
    {
        SRF_RETURN_DONE(funcctx);
    }
```

```
}
```

在 SQL 里声明这个函数的一个方法是：

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS '_filename_', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

另外一个方法是使用 OUT 参数：

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS '_filename_', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

请注意在这个方法里，函数的输出类型实际上是匿名的 `record` 类型。

参阅源码发布包里的contrib/tablefunc 获取更多有关返回集合的函数的例子。

35.9.10. 多态参数和返回类型

C 语言函数可以声明为接受和返回多态的类型 `anyelement`， `anyarray`， `anynonarray`， `anyenum` 和 `anyrange` 。 参阅Section 35.2.5获取有关多态函数的更详细解释。 如果函数参数或者返回类型定义为多态类型， 那么函数的作者就无法预先知道他将收到的参数，以及需要返回的数据。 在 `fmgr.h` 里有两个过程，可以让版本-1 的 C 函数知道它的参数的确切数据类型以及 它需要返回的数据类型。这两个过程叫 `get_fn_expr_rettype(FmgrInfo *flinfo)` 和 `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)` 。 它们返回结果或者参数的类型 OID, 如果这些信息不可获取，则返回 InvalidOid 。 结构 `flinfo` 通常是以 `fcinfo->flinfo` 进行访问的。 参数 `argnum` 是以 0 为基的。 `get_call_result_type` 也可以替代 `get_fn_expr_rettype` 。 还有 `get_fn_expr_variadic` 用于找出是否调用包含明确的 `VARIADIC` 关键字。 对于 `VARIADIC "any"` 函数是最有用的，正如下面所述。

比如，假设想写一个函数接受任意类型的一个元素，并且返回该类型的一个一维数组:

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

/* 获取提供的元素(要小心其为NULL的情况) */

    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

/* 维数是1 */

    ndims = 1;
/* 有1个元素 */

    dims[0] = 1;
/* 数组下界是1 */

    lbs[0] = 1;

/* 获取有关元素类型需要的信息 */

    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

 /* 然后制作数组 */

    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

下面的命令用SQL声明 `make_array` 函数：

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS '_DIRECTORY_/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

有一个变种多态性，仅适用于C语言函数：他们可以声明采取类型 `"any"` 的参数。（注意：这个类型名称必须是双引号， 因为它同时也是一个SQL的保留字）。类似于 `anyelement` 除了它并不限制不同 `"any"` 参数是相同类型，也没有帮助确定该函数的结果类型。一个C语言的函数也可以声明最后的参数为 `VARIADIC "any"` 。 这将匹配一个或多个任意类型的实参（不一定是相同的类型）。 这些参数不被收集到一个数组中如发生正常的可变参数函数； 他们会分别被传递到函数中。 `PG_NARGS()` 宏和 上面描述的方法必须被用来确定实际参数数目 以及使

用此功能时的类型。同时，这个函数的用户可能希望在函数调用中使用 `VARIADIC` 关键字， 以期望函数把数组元素看作单独的参数。函数本身必须实现 想要的操作，使用 `get_fn_expr_variadic` 之后 检测实际参数被标记为 `VARIADIC` 。

35.9.11. 转换函数

一些函数的调用可以在规划中基于函数的属性特性被简化。比如， `int4mul(n, 1)` 可简化为 `n` 。 为了定义函数-特定优化，写*transform function*并将其OID放入 基函数的 `pg_proc` 项的 `protransform` 字段中， 转换函数必须有SQL签名 `protransform(internal) RETURNS internal` 。 参数，其实 `FuncExpr *` 是代表调用基函数的一个虚拟节点。 如果表达式树的变换函数的研究证明简化的表达式树可以替代所有 可能的具体调用其表示建立并且返回简单的表达式。 否则，返回 `NULL` 指针(_不是_SQL null)。

我们不做任何保证， PostgreSQL不会调用这种情况下的主要函数以简化转换函数。 确保在简化的表达式以及实际调用主要函数之间的严格等价性。

当前，这个设施在SQL水平上不暴露给用户，出于安全考虑。因此只有实践中用于优化内置函数。

35.9.12. 共享内存和LWLocks

插件可能保留 LWLocks 并在服务器启动时分配共享内存。 插件的共享库必须通过指定shared_preload_libraries的方法预先加载。

```
void RequestAddinShmemSpace(int size)
```

共享内存可以通过在 `_PG_init` 函数中调用。

LWLocks通过调用进行预留：

```
void RequestAddinLWLocks(int n)
```

来自 `_PG_init` 。

为了避免可能的竞争条件，当连接并且初始化共享内存分配时， 每个后端应该使用LWLock `AddinShmemInitLock` ，如下所示：

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->mylockid = LWLockAssign();
        }
        LWLockRelease(AddinShmemInitLock);
}
```

35.9.13. 使用C++的可扩展性

尽管PostgreSQL后端以C写入，如果伴随这些准则， 在C++中写入扩展是可能的：

- 所有被后端访问的函数必须提供到后端的C接口； 这些C函数然后调用C++函数。比如， `extern C` 联系
- 使用合适的存储单元分配方法释放内存。比如，使用 `palloc()` 分配大部分后端内存， 因此使用 `pfree()` 释放它。在这种情况下使用C++ `delete` 将失败。
- 防止异常传播到C代码（使用捕获所有 `extern C` 函数的最高水平上的块）。 即使是C++代码没有明确地抛出异常，这是必要的，由于事件比如内存不足仍然可以抛出异常。任何异常必须被捕获，并且将适当的错误传递给C接口。如果可能的话，编译C++ `-fno-exceptions` 以完全消除异常；在这样的案例中， 你必须检查你的C++代码的错误，比如检查通过 `new()` 返回的NULL。
- 如果从C++代码中调用后端函数，确保C++调用堆栈中只包含纯旧的数据结构(POD)。 这是必要的因为后端错误产生一个遥远的 `longjmp()` ，不适当的展开与非-POD对象的C++调用堆栈。

总之，把C++代码放在与后端接口的 `extern C` 函数之后是最好的， 并且避免异常，内存以及调用堆栈泄露。

35.10. 用户定义聚集

在PostgreSQL里的聚集是用状态值 和状态转换函数表达的。 也就是说，聚集操作使用一个随着每个输入行被处理而变化的状态值。 要定义一个新的聚集函数，就要选择表示状态值的数据类型、状态初始值、状态转换函数。 该状态转换函数只是一个普通函数，也可以用于聚集的环境之外。 还可以声明一个最终处理函数， 用于对付期望的聚集结果不同于需要保留在状态值中数据的情况。

因此，除了被聚集用户看到的参数和结果数据类型外，还有一种内部状态值数据类型， 这种类型可能与参数和结果类型都不一样。

如果定义了一个不使用最终处理函数的聚集，那么聚集就是对每条记录的字段值进行函数计算。 `sum` (求和)是这类聚集的例子。它从零开始， 依次向"总和"状态值追加当前的记录值。比如，如果要把 `sum` 聚集用于复数， 只需要该数据类型的加法函数就行了。该聚集可以这样定义：

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);

SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

请注意，上述依赖于函数重载：有多个名为 `sum` 的聚集函数， 但是PostgreSQL能够正确选出作用于 `complex` 列类型的那个。

如果不存在非 NULL 输入值，上面的 `sum` 定义将返回零值(初始状态条件)。 要按照 SQL 标准的要求返回 NULL 只需忽略 `initcond` 段就可以实现(这样初始状态条件将变为 NULL)。 通常这也意味着 `sfunc` 需要检查 NULL 状态条件输入，不过对于 `sum` ， `max` ， `min` 这类的简单聚集来说，把第一个非空输入插入到状态值里面， 然后从第二个非空输入状态值开始使用转换函数就足够了。 如果初始条件是 NULL 并且转换函数被标记为"strict"(不能对 NULL 输入调用)， PostgreSQL就会自动处理这些内容。

另外一个"strict"转换函数的缺省特性是：当碰到一个 NULL 输入的时候， 前面一个状态值会被保留下来不做改动。这样，就忽略了 NULL 。 如果你希望对 NULL 输入进行其它处理，只需要别把你的转换函数定义为"strict"， 并在编写代码的时候测试 NULL 并做相应处理即可。

`avg` (平均)是聚集更复杂一点的例子。它需要两个运行时状态： 输入的总和以及输入的数量。最终结果是通过把两者相除得到的。 平均的典型实现是用一个数组做状态值。比如，内建的 `avg(float8)` 实现是这样的：

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

（`float8_accum` 要求一个三元素数组，而不是两元素， 因为它累积平方和和输入的总和和计数。这样它就可以在一些除了 `avg` 之外的聚集中使用了。）

聚集函数可以使用多态转换函数或者最终处理函数，这样，同一个函数就可以用于实现多个聚集。 参阅Section 35.2.5获取多态函数的解释。再进一步， 聚集函数本身可以用多态的基本类型和状态类型来声明， 这样就允许一个聚集定义用于多种输入数据类型。下面是一个多态聚集的例子：

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

这里，任意聚集调用的实际状态类型是和元素输入类型相同的数组类型。 聚集的特征是连接所有的输入到那个类型的数组里。 （注意： 内建的聚集 `array_agg` 支持相同的功能，并且有比这个定义更好的性能。）

下面的例子使用两个不同实际数据类型作为参数输出：

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

一个用 C 写的函数可以判断它是被当作一个聚集转换函数调用还是通过调用 `AggCheckCallContext` 作为最终的函数，例如：

```
if (AggCheckCallContext(fcinfo, NULL))
```

检查这个的一个原因是，在它对于一个转换函数为真的时候，左边的输入必须是一个临时的转换值， 因此可以安全地现场修改，而不用分配新的拷贝。参阅 `int8inc()` 的例子。 （这是函数里唯一可以修改输入的传递引用的地方。特别的， 聚集最终的函数不应该在任何情况下修改他们的输入， 因为在某些情况下它们将在相同的最终转换值下重复执行。）

更详细的信息请参考CREATE AGGREGATE命令。

35.11. 用户定义类型

正如Section 35.2所说，PostgreSQL 可以扩展为支持新的数据类型。本节描述如何定义新的基本类型， 这些类型是那些定义在SQL语言之下的数据类型。 创建一个新的基本类型要求实现函数在低层语言(通常是 C)的类型上操作。

本节的例子可以在源码发布中 `src/tutorial` 目录的 `complex.sql` 和 `complex.c` 里找到。参见同目录下的 `README` 文件获取关于如何运行例子的指示。

一个用户定义的类型总是有输入和输出函数。 这些函数决定该类型如何在字符串里出现(让用户输入和输出给用户)以及类型如何在内存里组织。 输入函数以一个以空(null)结尾的字符串为参数并且返回该类型的内部(内存里)的表现形式。 输出类型以该类型的内部表现形式为参数并且返回一个以空(null)结尾的字符串。 如果我们想做任何比简单存储它更多的事情，我们必须提供额外的函数来实现我们想要对这个类型的操作。

假设要定义一个 `complex` 类型来表示复数。通常， 选用下面的 C 结构来在内存里表现复数：

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

我们需要将它变成引用传递类型，因为它太大，不能放在一个单独的 `Datum` 值中。

对于该类型的外部表现形式，选择形如 `(x,y)` 的字符串。

输入输出函数通常并不难写，尤其是输出函数。但是，在定义你的外部(字符串)表现形式时，要注意你最后必须为该表现形式写一个完整而且健壮的分析器作为输入函数。比如：

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

输出函数可以简单的写成：

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

你应该把你的输入和输出函数做成互逆函数。 如果不这样做就可能在需要把数据输出来再加载回去时碰到很严重的问题， 当涉及浮点数时，这是非常普遍的问题。

另外，一个用户定义类型可以提供二进制输入和输出过程。二进制 I/O 通常更快， 但是没有文本 I/O 移植性好。因为对于文本 I/O 而言，完全由你来定义外部的二进制形式。 大多数内置的数据类型都尽可能提供一个与机器无关的二进制形式。对于 `complex`， 将把二进制 I/O 建立在 `float8` 的基础上：

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

一旦我们写好 I/O 函数并将其编译为共享库，就可以定义 SQL 中的 `complex` 类型。 我们首先声明其为 shell 类型：

```
CREATE TYPE complex;
```

这将作为一个占位符以允许在定义其 I/O 函数时引用该类型。现在我们定义其 I/O 函数：

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS '_filename_'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS '_filename_'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS '_filename_'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS '_filename_'
   LANGUAGE C IMMUTABLE STRICT;
```

最后，可以完整定义该数据类型：

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

当你定义一种新的基本类型时，PostgreSQL 自动提供对该类型的数组支持。因为历史原因，数组类型的类型名是与该类型同名的字符串前面加个下划线字符(_)。

一旦数据类型存在，就可以声明额外的函数以提供在该数据类型上的操作， 然后就可以在这些函数上定义操作符。如果需要， 还可以创建操作符类支持该数据类型的索引。这些将在后面的章节介绍。

如果你的数据类型的大小是变化的(内部形式)，那么你应该把它们标记为可 TOAST的(参阅Section 58.2)。 即使数据总是太小以至于被压缩或存储在外部你也应该这样做，因为 TOAST 可以通过减少头开销在小数据上节约空间。

要做到这一点，该类型的内部形式必需遵循变长数据内部形式的标准布局： 头四个字节必需是一个从来没有直接访问的 `char[4]` 字段（通常叫 `vl_len_` ）。你必须使用 `SET_VARSIZE()` 来存储这个字段里的数据的长度，使用 `VARSIZE()` 来取回这个长度。 在该类型上操作的 C 函数必须通过使用 `PG_DETOAST_DATUM` 小心地解开它们处理的任何"烘烤"过的数值(这些细节通常都可以通过定义类型相关的 `GETARG_DATATYPE_P` 宏掩盖)。最后， 在使用 `CREATE TYPE` 命令的时候，声明内部长度为 `variable` 并且选择恰当的存储选项。

如果对齐是不重要的（不管是只是对于一个特定的函数还是因为数据类型指定字节对齐） 那么避免 `PG_DETOAST_DATUM` 的一些开销是可能的。 你可以使用 `PG_DETOAST_DATUM_PACKED` 代替（通常通过定义一个 `GETARG_DATATYPE_PP` 宏指令），并且使用宏指令 `VARSIZE_ANY_EXHDR`

和 `VARDATA_ANY` 来使用一个潜在封装的数据。 另外，由这些宏指令返回的数据是不对齐的，即使数据类型定义声明了一个对齐。 如果对齐是重要的，那么你必须通过正规的 `PG_DETOAST_DATUM` 接口。

> **Note:** 旧代码经常声明 `vl_len_` 为一个 `int32` 字段而不是 `char[4]` 。 只要结构体的定义中其他字段至少有 `int32` 个队列的话这么做是可以的。 但是当使用一个潜在的对齐数据时，使用这样一个结构体定义是危险的； 编译器可能把它当做一个许可来假设数据实际上已经对齐，导致在架构上的核心转储严格对齐。

更多细节请参阅CREATE TYPE命令。

35.12. 用户定义操作符

每个操作符都是对真正干活的对应函数的"语义修饰"；所以你在创建操作符之前必须先创建对应的函数。不过，一个操作符也并不仅仅是语义修饰，因为它还带着可以帮助查询规划器优化查询使用该操作符的附加信息。 下一节将用于解释这些附加信息。

PostgreSQL支持左目、右目、双目操作符。操作符可以重载； 也就是说，同一个操作符名可以用于不同数目和类型的操作数的操作符。在执行一个查询的时候， 系统从提供的操作数的数量和类型上判断需要调用哪个操作符。

下面是一个创建用于两个复数相加的操作符的例子。假设已经创建了`complex` (见节Section 35.11)类型的定义。首先需要做相加工作的函数；然后就可以定义操作符：

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS '_filename_', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

现在可以执行像下面这样的查询：

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

在这里已经演示了如何创建双目操作符。要创建单目操作符，只需要省略`leftarg`或者`rightarg`即可。只有`procedure`子句和参数(argument)子句是`CREATE OPERATOR`里需要的条目。例子里演示的`commutator` 子句是一个给查询优化器的可选暗示。关于`commutator` 和其它优化器提示的详细信息在下节给出。

35.13. 操作符优化信息

PostgreSQL的操作符定义可以包括几个可选的子句， 这些子句告诉系统一些关于该操作符特性的有用信息。在可能的情况下，都应该提供这些子句， 因为它们可能为使用这个操作符的查询带来可观的速度提升。不过要注意如果你声明了这些子句， 就必须确保它们是正确的！对优化子句的错误使用将导致减慢查询速度、微小的输出错误、 或者其它糟糕事情。如果你对这些事情不确定的话，可以总是忽略优化子句；唯一的后果就是查询可能运行的慢一些。

附加的优化子句可能在今后的PostgreSQL版本里增加。 这里描述的都是9.3.1版本可以理解的。

35.13.1. `COMMUTATOR`

如果提供了`COMMUTATOR`子句，则命名一个操作符是被定义的操作符的交换符。 如果有两个操作符 A, B， 对于任何可能的输入数值 x, y 都有(x A y)等于(y B x)， 那么就说 A 是 B 的交换符，同样 B 也是 A 的交换符。例如，操作符`<`和`>` 对于所使用的一定的数据类型通常都是对方的交换符，而操作符`+`通常是它自身的交换符。 但是操作符`-`通常没有交换符。

交换操作符的左操作数与右操作数类型必须相同。所以PostgreSQL 所需要的只是一个交换符操作符的名称用以查找该交换符，那也是`COMMUTATOR` 子句里所需要的唯一的东西。

给那些会在索引和连接子句里面使用的操作符提供交换符是非常关键的， 因为这样就允许查询优化器"移动"这样的子句，形成所需要的不同的规划类型的形式。 比如，考虑一个有类似`tab1.x = tab2.y`的 WHERE 子句的查询， 这里`tab1.x`和`tab2.y`是用户定义类型，并且假设`tab2.y`上面有索引。 除非优化器知道如何在`tab2.y = tab1.x`周围四处移动该子句，否则它不能生成索引扫描， 因为索引扫描机制期望看到索引字段在给出的操作符左边。PostgreSQL 不会简单地假设这是一个合法的转换， `=`的创建者必须声明这是有效的， 方法是给这个操作符标记交换器信息。

当你定义一个自交换的操作符时，你简单的定义它就可以了。当你定义一对交换符操作符时， 事情就有一点棘手：怎样定义一个操作符的交换符指向另一个你还没有定义的操作符呢? 对这个问题有两个解决方法：

- 一个方法是省略你定义的第一个操作符的`COMMUTATOR`子句，然后在第二个操作符的定义里提供一个。 因为PostgreSQL知道交换操作符是成对出现的， 所以当它看到第二个定义时它会自动折回并填充第一个定义里空缺的`COMMUTATOR`子句。

- 另一个更直接的方法是在两个定义里面都包含`COMMUTATOR`子句。当PostgreSQL 处理第一个定义并意识到`COMMUTATOR`指向一个不存在的操作符时会在系统表里为该操作符记录一个虚拟记录。 这个虚拟的记录只有操作符名，左和右操作数类型以及结果类型是有效

的，因为这些是到目前为止 PostgreSQL可以推导出来的东西。第一个操作符类记录将和这个虚拟记录连接。 稍后，当你定义第二个操作符时，系统将用来自第二个操作符的信息更新该虚拟记录。 如果你试图在虚拟操作符被填充之前使用它，你将只能收到一条错误信息。

35.13.2. `NEGATOR`

如果提供了 `NEGATOR` 子句，则命名一个操作符是被定义的操作符的否定符。 如果有两个都返回布尔变量的操作符 A 和 B ，对任何可能的输入 x 和 y ，都有 (x A y) 等于 NOT(x B y)，那么说 A 是 B 的否定符。当然 B 也是 A 的否定符。例如， `<` 和 `>=` 对大多数数据类型是一对否定符。一个操作符不可能是它自身的否定符。

不像交换符，一对单目操作符可以互为否定符；那就意味着对于所有的 x 都有 (A x) 等于 NOT(B x) ， 或者类似的右目操作符的这种情况。

一个操作符的否定符必须有与正定义的操作符本身一样的左和/或右操作数类型， 所以就像 `COMMUTATOR` 一样，只有操作符名需要在 `NEGATOR` 子句里面给出。

提供否定符对查询优化器是非常有帮助的，因为这样就允许像 `NOT (x = y)` 这样的表达式简化成 `x <> y` 。这样的情况比你想像的要频繁的多， 因为 `NOT` 操作可能因为其它的重排列而被引入。

否定符对可以用上面交换符对中解释的相同的方法来定义。

35.13.3. `RESTRICT`

如果提供了 `RESTRICT` 子句，则为操作符命名一个选择性限制计算函数(注意这里是一个函数名， 而不是一个操作符名)。 `RESTRICT` 子句只是对返回 `boolean` 变量的双目操作符有意义。选择性限制计算符的概念是猜测一个表中所有行的哪一部分对于目前的操作符和特定的常量将满足一个像下面这样形式的 `WHERE` 条件子句。

```
column OP constant
```

它可以给出这种类型的 `WHERE` 子句可以删除多少行的一个概念，这将帮助优化器进行优化。你可能会说，如果该常量(constant)在左边怎么办?哦，那是 `COMMUTATOR` 干的事...

书写新的选择性限制计算函数远远超出了本章的范围，不过很幸运的是， 通常你对自己的操作符只需要使用系统标准的计算器之一就行了。下面是一些标准限制计算器：

`eqsel` 用于 `=`
`neqsel` 用于 `<>`
`scalarltsel` 用于 `<` 或 `<=`
`scalargtsel` 用于 `>` 或 `>=`

这些都是分类，看起来有点奇怪，不过如果你仔细想想，就会觉得有道理。 `=` 大多将只接受表中的一小部分行； `<>` 大多将拒绝一小部分行。 `<` 接受的行取决于给出的常量落在表的该列数据值的哪一个范围里 (该值碰巧是 `ANALYZE` 收集并且提供给选择性计算器的信息)。 `<=` 在同样的常量时会接受比 `<` 略微大一些的行，不过它们也非常接近，几乎不值得区别开来，尤其是无论如何也比做盲猜好得多。 类似的情况也适用于 `>` 和 `>=` 。

你可能常习惯于把 `eqsel` 或 `neqsel` 用于那些非常高或者非常低选择性的操作符， 即使它们并非真正相等或者不相等。例如，基于只会匹配整个表中一小部分记录的假设，几何操作符约等于就使用 `eqsel` 。

你可以把 `scalarltsel` 和 `scalargtsel` 用于比较那些为进行范围比较被转化为数字尺度后有明显意义的数据类型。如果可能， 把该数据类型增加到可以被 `src/backend/utils/adt/selfuncs.c` 文件里的 `convert_to_scalar()` 函数理解的部分。最终，这个过程将被放到由 `pg_type` 表里的一个列标识的每种类型一个的函数代替，不过目前还没有这么做。如果你没有做这些，系统仍然能工作， 不过优化器的估计不会像想像的那么好。

在 `src/backend/utils/adt/geo_selfuncs.c` 里还有为几何操作符设计的额外选择性评估函数： `areasel` , `positionsel` , `contsel` 。目前， 它们都只是存根，但是你还是可以使用(最后是改良)它们。

35.13.4. `JOIN`

如果提供了 `JOIN` 子句，则为操作符命名一个连接选择性计算器函数(是函数名， 不是操作符名)。 `JOIN` 子句只是对返回 `boolean` 的双目操作符有意义。 一个连接选择性计算器后面的概念是猜测一对表上的哪部分行对目前的操作符将满足下面形式的 `WHERE` 子句的条件：

```
table1.column1 OP table2.column2
```

和 `RESTRICT` 子句一样， 这些很有可能帮助优化器用最少的处理勾画出要采取可能的连接顺序中的哪一个。

和前面一样，本节不会试图解释如何书写一个连接选择性计算器函数， 但是会建议你尽可能使用一个标准的计算器：

| `eqjoinsel` 用于 `=` || `neqjoinsel` 用于 `<>` || `scalarltjoinsel` 用于 `<` 或 `<=` || `scalargtjoinsel` 用于 `>` 或 `>=` || `areajoinsel` 用于基于面积的二维比较 || `positionjoinsel` 用于基于位置的二维比较 || `contjoinsel` 用于基于包含的二维比较 |

35.13.5. `HASHES`

如果出现了 `HASHES` 子句，则告诉系统对于一个基于此操作符的连接可以使用 Hash 连接。 `HASHES` 只对返回 `boolean` 的双目操作符有意义， 并且实际上该操作符最好是对某种数据类型的相等操作符。

Hash 连接的假设是：对于一对散列到同样的 Hash 代码的左和右操作数值，该连接操作符只能返回真。 如果两个值被放到不同的 Hash 桶里，连接将根本不比较它们，隐含地意味着连接操作符的结果一定是假。 所以对于不代表相等的操作符，声明 `HASHES` 是没有意义的。 在大多数情况下，支持两端接受同样数据类型的操作符是唯一可行的。然而， 有时为两个或更多的数据类型设计兼容的hash函数也是可能的；也就是，函数将为"相等的" 值产生相同的hash代码，即使值有不同的代表。例如，当哈希整数有不容的宽度时，排列这个属性是非常简单。

要标记为 `HASHES` ，连接操作符必须出现在一个 Hash 索引操作符类中。 在创建操作符时并不强制这样，因为引用操作符类不可能还存在。 但是企图在 Hash 连接中使用尚不存在的操作符类将在运行时导致失败。 系统需要操作符类根据操作符的输入数据类型确定特定于该数据类型的 Hash 函数。当然， 你必须在创建操作符类之前首先提供合适的 Hash 函数。

在编写 Hash 函数时必须小心，因为有一些硬件相关的因素会导致错误。比如， 如果你的数据类型是一个存在间隙的结构体，你就不能简单的将其传递给某个 `hash_any` 函数。除非你的其它操作符能够确保这些间隙总是零(这是建议的策略)。 另一个例子是在符合IEEE浮点标准的机器上，负零和正零是不同的值(不同的位模式)， 但是它们被定义为比较相等。如果一个浮点值可能包含负零， 那么必须使用额外的步骤来确保产生和正零相同的 Hash 值。

一个可 Hash 连接的操作符必须有一个在相同操作符类中的交换符（如果两个操作符数据类型相同则是它本身， 如果不同则是一个相关的相等操作符）。如果不是这样，当使用操作符时会发生规划器错误。同样， 一个hash操作符类支持多种数据类型以为数据类型的每种结合提供相等操作符是一个好主意（但不是严格要求）；这允许更好的优化。

> **Note:** 在一个可 Hash 连接的操作符下层的函数必须标明 immutable 或 stable 。如果它是 volatile ， 那么系统将从不在 Hash 连接中使用这些操作符。
>
> **Note:** 如果一个可 Hash 连接的操作符有一个下层函数标记为严格的(strict)，那么该函数必须完整： 也就是说，对于任何非 NULL 输入， 它应该返回 TRUE 或 FALSE ， 但绝不能是 NULL 。 如果不遵循这个规则， `IN` 操作的 Hash 优化可能会生成错误的结果。 特别是根据规范正确答案是 NULL 的时候， `IN` 可能会返回 FALSE ； 或者它可能生成一个错误，抱怨说它对 NULL 结果没有思想准备。

35.13.6. `MERGES`

如果出现了 `MERGES` 子句，则告诉系统对基于目前操作符的连接可以使用融合连接方法。 `MERGES` 只是对返回 `boolean` 的双目操作符有意义， 实际上这个操作符对于某些数据类型或者某对数据类型必须表示相等。

融合连接是以这样的概念为基础的：对左边和右边的表进行排序，然后并发地扫描它们。所以， 两种数据类型都必须是能够完全排序的，并且连接操作符必须只对那些落在排序顺序中的 "某个位置"的数值对成功。实际上这意味着连接操作符必须表现得像等于。 但是可以对两种不同数据类型进行融合连接(只要他们逻辑相等即可)。例如， `smallint` 对 `integer` 的相等操作符是可以用融合连接的。 只需要可以把两种数据类型排列成逻辑可比序列的排序操作符即可。

要标记为 `MERGES` ，连接操作符必须作为 `btree` 索引操作符类的一个相等的成员出现。 在创建操作符时并不强制这么做，因为引用操作符类不可能还存在。 但是操作符不会被实际用于融合连接，除非可以找到一个匹配操作符类。 `MERGES` 标志因此作为一个对规划器的提示，查找一个匹配的操作符类是值得的。

可融合连接的相等操作符必须有一个在同一个操作符类中的交换符 （如果两个操作数数据类型相同则是它自身，如果不同则是一个相关的相等操作符）。 如果不是这样，当使用操作符时会发生规划器错误。同样， 一个 `btree` 操作符类支持多种数据类型以为数据类型的每种结合提供相等操作符是一个好主意 （但不是严格要求）；这允许更好的优化。

> **Note:** 在一个可融合连接操作符下层的函数必须标记为永久(immutable)或者稳定(stable)。 如果它是易失的(volatile)，那么系统将从不在融合连接中使用这些操作符。

35.14. 扩展索引接口

到目前为止描述的过程可以让你定义一个新类型、新函数、新操作符。但是，还不能在一个新数据类型的字段上面定义一个索引。为了达到这个目的，必须为新数据类型定义一个操作符类。下面将使用一个真实的例子来描述操作符类：一个用于 B-tree 访问方法的新操作符类，它保存复数并按照绝对值递增的顺序排序。

操作符类可以分类到操作符族，用以显示语义兼容的类之间的关系。当只包含一个数据类型时，一个操作符类就足够了，所以我们首先关注这种情况，然后再转到操作符族。

35.14.1. 索引方法和操作符类

`pg_am` 表为每个索引方法(内部称作访问方法)都包含一条记录。对表的普通访问方法支持内建于PostgreSQL，但所有索引方法在 `pg_am` 里都有描述。可以通过定义要求的接口过程并在 `pg_am` 里创建一个新行的办法增加一个索引访问方法，不过这些些远远超出了本章的内容(参阅Chapter 54)。

一个索引方法的过程并不直接知道任何该索引方法将要操作的数据类型的信息。而是操作符类表明索引方法在操作特定数据类型的时候需要使用的操作集合。操作符类的名称的由来是因为它们声明是一种索引可以使用的 `WHERE` 子句的操作符集(也就是可以转化成一个索引扫描条件)。一个操作符类也可以声明一些索引方法需要的内部操作的支持过程，但是它们并不直接和可以与索引一起使用的 `WHERE` 子句操作符相关。

可以为同一个数据类型和索引方法定义多个操作符类。这么做的结果是，可以为一种数据类型定义多套索引语义。比如，一个 B-tree 索引要求为它操作的每种数据类型定义一个排序顺序。对于一个复数数据类型而言，有一个通过复数绝对值对数据排序的 B-tree 操作符类可能会有用，还有一个是用实部排序，等等。通常其中一个操作符类会被认为最常用的，并且被标记为该数据类型和索引方法的缺省操作符类。

同样的操作符类名字可以用于多种不同的索引方法(比如 B-tree 和 Hash 访问方法都有叫 `int4_ops` 的操作符类)，但是每个这样的表都是一个独立的实体，必须分别定义。。

35.14.2. 索引方法策略

和一种操作符类相关联的操作符是通过"策略号"标识的，策略号用于标识每种操作符在它的操作符类环境里的语义。比如，B-tree 对键字有严格的排序要求，小于到大于，因此，像"小于"和"大于或等于"这样的操作符都是 B-tree 所感兴趣的。因为PostgreSQL允许用户定义操

作符，PostgreSQL 无法仅通过查看操作符的名字(比如 < 或 >=)就明白它进行的比较是什么。 实际上，索引方法定义了一套"策略"，它可以看作一般性的操作符。 每种操作符类显示对于特定数据类型而言，是哪种实际操作符对应每种策略以及解释索引的语义。

B-tree 索引定义了五种策略。在Table 35-2中显示。

Table 35-2. B-tree 策略

操作	策略号
小于	1
小于或等于	2
等于	3
大于或等于	4
大于	5

Hash 索引只支持平等的比较，因此它们只定义了一个策略， 在Table 35-3里显示。

Table 35-3. Hash 策略

操作	策略号
等于	1

GiST 索引甚至更加灵活：它们根本就没有固定的策略集。实际上， 是每个特定 GiST 操作符类的"一致性"支持过程解释策略号是什么样子。作为示例， 有几个内置的 GiST 索引操作符类索引二维几何对象，提供Table 35-4 中所示的"R-tree"策略。其中的四个是两维测试(重叠、相同、包含、包含于)；四个只考虑 x 坐标、四个对 y 坐标进行同样测试。

Table 35-4. GiST 两维"R-tree"策略

操作	策略号
严格地在...左边	1
不扩展到...右边	2
重叠	3
不延伸到...左边	4
严格地在...右边	5
相同	6
包含	7
包含于	8
不扩展到...上面	9
严格地在...下面	10
严格地在...上面	11
不扩展到...下面	12

SP-GiST索引在灵活性方面与 GiST 索引类似：它们都没有一个固定的策略集， 而是由每个操作符类的支持过程根据操作符类的定义来解释策略号。作为示例， Table 35-5显示了内置的点操作符类使用的策略号。

Table 35-5. SP-GiST 点策略

操作	策略号
严格在左边	1
严格在右边	5
相同	6
包含	8
严格在下面	10
严格在上面	11

GIN 索引与 GiST 索引和SP-GiST 索引类似：它们都没有一个固定的策略集， 而是由每个操作符类的支持过程根据操作符类的定义来解释策略号。作为示例， Table 35-6显示了内置的数组操作符类使用的策略号。

Table 35-6. GIN 数组策略

操作	策略号
重叠	1
包含	2
包含于	3
相等	4

请注意，所有上述操作符都返回布尔值。实际上，所有定义为索引方法搜索操作符的操作符都必须返回 `boolean` 类型，因为它们必须出现在一个 `WHERE` 子句的顶层， 这样才能被一个索引使用。（某些索引访问方法也支持顺序操作符， 它们通常不反悔Boolean值；这个特征在Section 35.14.7里讨论。）

35.14.3. 索引方法支持过程

有时候，策略的信息还不足以让系统决定如何使用某个索引。在实际中， 索引方法需要附加的一些过程来保证正常工作。例如， B-tree 索引方法必须能够比较两个键字以决定其中一个是大于、等于、还是小于另外一个。 类似的还有 Hash 索引方法必须能够在键值上计算散列值。 这些操作和 SQL 命令条件里使用的操作符并不对应；它们是在内部被索引方法使用的管理过程。

就像策略一样，操作符类声明在一定的数据类型和语义解释的条件下， 哪个特定函数对应这些角色中的哪一个。索引方法声明它需要的函数集， 而操作符类通过给它们赋予通过索引方法指定的"支持函数编号"来标识要正确使用的函数。

B-tree 需要一个支持函数，并且允许在操作符类作者的选项中提供第二个， 就像Table 35-7里显示的那样。

Table 35-7. B-tree 支持函数

函数	支持号
比较两个键字并且返回一个小于、等于、大于零的整数，标识第一个键字小于、等于、大于第二个键字。	1
返回C-callable排序支持函数的地址，记录在 `utils/sortsupport.h` （可选）	2

Hash 索引也需要一个支持函数，在Table 35-8里显示。

Table 35-8. Hash 支持函数

函数	支持号
为一个键字计算散列值	1

GiST 索引需要七种支持函数，和一个可选的函数，在Table 35-9里显示。 （更多信息请参考Chapter 55。）

Table 35-9. GiST 支持函数

函数	描述	支持号
`consistent`	检测键是否满足查询限定符	1
`union`	计算一套键的联合	2
`compress`	计算已索引键或值的压缩结果	3
`decompress`	计算已压缩键的解压结果	4
`penalty`	计算使用给定的子树的键向子树中插入新键的性能恶化(penalty)	5
`picksplit`	检测页面中的哪个项将被移动到新页面并为结果页计算联合键	6
`equal`	比较两个键并在相等时返回真	7
`distance`	确定键到查询值的距离（可选）	8

SP-GiST索引需要五种支持函数，显示在Table 35-10中。 （更多信息请参阅Chapter 56。）

Table 35-10. SP-GiST 支持函数

函数	描述	支持号
`config`	提供操作符类的基本信息	1
`choose`	确定如何将一个新值插入一个内在的元组	2
`picksplit`	确定如何分区一组值	3
`inner_consistent`	确定哪个子分区需要为一个查询搜索	4
`leaf_consistent`	确定哪个键满足查询条件	5

GIN 索引需要四种支持函数，和一个可选的函数，在Table 35-11里显示。 （更多信息请参阅Chapter 57。）

Table 35-11. GIN 支持函数

函数	描述	支持号
`compare`	比较两个键并返回一个小于、等于、大于零的整数，标识第一个键小于、等于、大于第二个键。	1
`extractValue`	从将被索引的值中抽取键	2
`extractQuery`	从查询条件中抽取键	3
`consistent`	检测值是否匹配查询条件	4
`comparePartial`	比较部分来自查询的键和来自索引的键，并返回一个小于、等于、大于零的整数，标识是否GIN应该忽略这个索引项，将这个项视为一个匹配，或停止索引扫描（可选）。	5

和搜索操作符不同，支持函数返回特定索引方法预期的数据类型，比如在 B-tree 的情况下，返回一个有符号整数。每个支持函数的参数的数字和类型也取决于索引方法。 对于B-tree和hash的比较，散列支持函数接受相同的输入数据类型，同样操作符也包含在操作符类里， 但是大多数GiST, SP-GiST, 和GIN 支持函数不是这样的。

35.14.4. 例子

既然已经了解了这些概念，那么现在就来看一个创建新操作符类的例子。你可以在源代码的 `src/tutorial/complex.c` 和 `src/tutorial/complex.sql` 中找到这里讲述的例子。操作符类封装了那些以绝对值顺序对复数排序的操作符，这样就可以选择 `complex_abs_ops` 这个名字。首先，需要一个操作符集合。 用于定义操作符的过程已经在Section 35.12讨论过了。对这个用于 B-tree 的操作符类， 需要的操作符是：

- 绝对值 小于 (策略 1)
- 绝对值 小于等于 (策略 2)
- 绝对值 等于 (策略 3)
- 绝对值 大于等于 (策略 4)
- 绝对值 大于 (策略 5)

定义一组相关的比较操作符最不容易出错的方法是首先写出 B-tree 比较支持函数， 然后再写出其它封装了支持函数的单行函数。这就减少了某些情况下导致不一致结果的机会。 根据这个指引，首先写出：

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

现在，小于函数看起来像这样：

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

其它四个函数的不同之处仅在它们如何将内部函数的结果与零比较。

下一步，基于 SQL 函数声明函数和操作符：

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS '_filename_', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

指定正确的交换器和"非"操作符以及适当的限制和连接选择性函数都是非常重要的， 否则优化器将无法有效地利用索引。请注意，小于、等于、 大于三种情况下应该使用不同的选择性函数。

其它几个值得注意的问题：

- 只可以有一个已命名操作符 `=` 把 `complex` 类型做为其两个操作数。 这种情况下没有其它用于 `complex` 的 `=` 操作符， 但是如果要制作一个实用的数据类型，可能需要 `=` 做为复数的普通等于操作。 这种情况下，可能需要使用一些其它操作符名称来命名 `complex_abs_eq` 。

- 尽管PostgreSQL可以处理 SQL 名字相同的函数， 只要它们的输入数据类型不同，而 C 只能处理一个具有给定名称的全局过程。 因此不能把 C 函数命名为像 `abs_eq` 这样简单的名字。 通常在 C 函数名里面包含数据类型名称是一个好习惯，这样就不会和用于其它数据类型的函数冲突。
- 可以制作名为 `abs_eq` 的 SQL 函数，依靠PostgreSQL 通过输入数据类型的不同来区分任何其它同名 SQL 函数。为了令例子简单， 做的函数在 C 层次和 SQL 层次都有相同的名称。

下一步是注册 B-tree 需要的"支持过程"。实现这个例子的 C 代码在包含操作符函数的同一个文件中， 下面是定义函数的方法：

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS '_filename_'
    LANGUAGE C IMMUTABLE STRICT;
```

既然已经有了需要的操作符和支持过程，就可以最后创建这个操作符类了：

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

这样就完成了！现在可以在一个 `complex` 列上创建和使用 B-tree 索引了。

可以把操作符记录写得更冗余一些，像：

```
        OPERATOR        1       < (complex, complex) ,
```

但是如果该操作符接受的数据类型是定义的操作符类处理的东西，那就没必要这么做。

上面的例子假设你想把这个新操作符类作为 `complex` 数据类型的缺省 B-tree 操作符类。 如果你不想这么做，只要去掉关键字 `DEFAULT` 即可。

35.14.5. 操作符类和操作符族

到目前为止我们都隐含的假定一个操作符类只能处理一种数据类型。 虽然每个索引字段都只能是单独一种数据类型， 但是使用索引操作符来比较一个已索引字段和一个不同类型的值常常很有用处。 如果有用于与一个操作符类连接的交叉数据类型操作符，通常是其他数据类型

有他自己的相关的操作符类。 这对于在相关的类之间明确的建立连接是有帮助的，因为这可以帮助规划器优化SQL查询 （尤其对于B-tree操作符类，因为规划器包含大量的关于如果处理这些问题的信息）。

为了处理这种需求，PostgreSQL使用操作符族的概念。一个操作符族包含一个或多个操作符类， 也可以包含可索引的操作符和对应的支持函数，作为一个整体属于这个族，但不是这个族中的任何一个类。 我们说这样的操作符和函数是"松散"在族里的，而不是被绑定到一个特定的类。 通常每个操作符类包含一个数据类型操作符，而交叉数据类型操作符是散落在族里的。

所有在一个操作符族里的操作符和函数必须有兼容的语法，兼容性需求是通过索引方法设置的。 你可能想知道为什么费心的挑选出特别的族的子集作为操作符类；并且甚至为了多种目的类的区分是不相关的，族只对分组感兴趣。 定义操作符类的原因是指定多少族需要支持任何特定的索引。如果有一个索引使用一个操作符类， 然后操作符类不能在不删除索引的情况下被删除，但是操作符族的其他部分， 即其他操作符类和松散的操作符可以被删除。因此，一个操作符类应该被指定包含最少的操作符和函数， 应该是在一个特定数据类型上索引工作所需要的适当的操作符和函数， 然后相关的但非重要的操作符可以作为松散的操作符族成员添加。

作为一个例子，PostgreSQL有一个内置的B-tree操作符族 `integer_ops` ， 它包含操作符类 `int8_ops` ， `int4_ops` 和 `int2_ops` ， 分别对 `bigint` (`int8`), `integer` (`int4`), 和 `smallint` (`int2`) 字段索引。也包含交叉数据类型比较操作符，允许其中的任意两种类型进行比较， 所以任意其中一种类型上的索引可以使用其他类型的比较值被搜索到。 族可以通过下面的定义复制：

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  <!--
  -- standard int8 comparisons
  -->
  -- 标准 int8 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  <!--
  -- standard int4 comparisons
  -->
  -- 标准 int4 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ;
```

```
CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  <!--
  -- standard int2 comparisons
  -->
  --标准 int2 比较
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  <!--
  -- cross-type comparisons int8 vs int2
  -->
  -- 交叉类型比较 int8 对 int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  <!--
  -- cross-type comparisons int8 vs int4
  -->
  -- 交叉类型比较 int8 对 int4
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,

  <!--
  -- cross-type comparisons int4 vs int2
  -->
  -- 交叉类型比较 int4 对 int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  <!--
  -- cross-type comparisons int4 vs int8
  -->
  -- 交叉类型比较 int4 对 int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,

  <!--
  -- cross-type comparisons int2 vs int8
  -->
  -- 交叉类型比较 int2 对 int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
  OPERATOR 3 = (int2, int8) ,
  OPERATOR 4 >= (int2, int8) ,
  OPERATOR 5 > (int2, int8) ,
  FUNCTION 1 btint28cmp(int2, int8) ,

  <!--
  -- cross-type comparisons int2 vs int4
```

```
    -->
    -- 交叉类型比较 int2 对 int4
    OPERATOR 1 < (int2, int4) ,
    OPERATOR 2 <= (int2, int4) ,
    OPERATOR 3 = (int2, int4) ,
    OPERATOR 4 >= (int2, int4) ,
    OPERATOR 5 > (int2, int4) ,
    FUNCTION 1 btint24cmp(int2, int4) ;
```

需要注意的是，这里的定义"重载"了操作符策略和支持函数号：每个号在族内多次发生。 只要每个数字的实例都有不同输入数据类型就都是允许的。输入类型都等于操作符类的输入类型的实例是主操作符， 并且支持该操作符类的函数，在大多数情况下应该被声明为操作符类的一部分，而不是该族内的松散成员。

在一个B-tree操作符族内，所有的操作符都必须适当的排序，意味着传递法保存所有该族支持的数据类型： "if A = B and B = C, then A = C",和"if A < B and B < C, then A < C"。 此外，代表操作符族的类型间的隐式的或二进制强制转换必须不能改变相关的排序次序。 族内的每个操作符必须有一个支持的函数，这个函数有和操作符相同的两个输入数据类型。 建议一个族是完整的，也就是，对于每个数据类型的组合，所有的操作符都包括了。 每个操作符类应该只包含非交叉类型操作符和它的数据类型的支持函数。

要建立一个多数据类型散列操作符族，必须为每个该族支持的数据类型创建兼容的散列支持函数。 这里的兼容意味着函数保证对两个通过族的相等运算符认为相等的两个值返回相同的散列码， 甚至两个值属于不容的类型时也是。当类型有不同的物理表示时这通常是很难完成的， 但是在某些情况下是可以做到的。更多的，通过隐式的或二进制强制转换， 转换一个操作符族中的数据类型的值到另一个同样在操作符族中的数据类型， 必须不能改变计算散列值。注意每个数据类型只有一个支持函数，而不是每个相等操作符。 建议一个族是完整的，也就是，对于每个数据类型的组合都提供一个相等操作符。 每个操作符类应该只包含非交叉类型相等操作符和它的数据类型的支持函数。

GiST, SP-GiST, 和 GIN索引对于交叉数据类型操作符没有任何明确的概念。 支持的操作符集对于可以处理的给定的操作符类只是主要的支持函数。

> **Note:** 在PostgreSQL 8.3之前，没有操作符族的概念， 因此任何试图和索引一起使用的交叉数据类型操作符必须直接绑定到索引的操作符类里面。 虽然这种方法仍然有效，但是已经弃用了，因为它使得索引的依赖太过广泛， 并且因为当数据类型都有操作符在相同的操作符族内时，规划器可以更有效的处理交叉数据类型比较。

35.14.6. 操作符类的系统相关性

除了是否可以用于索引外，PostgreSQL还有多种途径使用操作符类来推断操作符性质。 因此，即使并不打算为你自定义的数据类型在任何字段上建立索引，你可能还是希望创建操作符类。

特别是诸如 `ORDER BY` 和 `DISTINCT` 之类需要对值进行比较和排序的 SQL 特性。 要在自定义的数据类型上实现这些特性，PostgreSQL 将会为该类型查找默认的 B-tree 操作符类。该操作符类中的"equals" 成员为 `GROUP BY` 和 `DISTINCT` 定义了相等的概念， 同时操作符类的排序顺序定义了默认的 `ORDER BY` 排序。

用户自定义类型数组的比较同样也依赖于默认 B-tree 操作符类定义的语意。

如果对于某个数据类型不存在默认 B-tree 操作符类，那么系统将会自动寻找默认的 Hash 操作符类。 但因为 Hash 操作符类仅仅提供相等比较，所以在实践中它仅能用于数组的相等性测试。

如果某个数据类型不存在任何缺省操作符类，你就会在使用该 SQL 特性时得到一个类似"could not identify an ordering operator"的错误。

> **Note:** PostgreSQL 7.4 以前，排序和分组操作隐含使用名为 `=` , `<` , `>` 的操作符。新的依赖默认操作符类的行为避免了对任何特定操作符名的行为的假定。

另一点重要的是一个在hash操作符族中的操作符是hash连接，hash聚合和相关优化的候选。hash操作符族在这里是重要的，因为它标志要使用的hash函数。

35.14.7. 排序操作符

一些索引访问方法（当前只有GIST）支持排序操作符的概念。 我们当前已经讨论过的是搜索操作符。搜索操作符是可以搜索索引找到所有满足 `WHERE` `_indexed_column_` `_operator_` `_constant_` 的行。 注意，不保证将要返回的匹配行的顺序。相反的，排序操作符不限制要返回的行集， 但是决定它们的顺序。排序操作符是可以扫描索引以 `ORDER BY` `_indexed_column_` `_operator_` `_constant_` 的顺序返回行。 这种方式定义排序操作符的原因是支持最近搜索，如果操作符是测量距离。例如，像这样的查询

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

找到一个距离给定目标点最近的点。在location字段上的GIST索引可以有效地做到这点， 因为 `<->` 是一个排序操作符。

当搜索操作符必须返回布尔结果时，排序操作符通常返回一些其他类型，如float或numeric。 这种类型通常不同于被索引的类型。为了避免关于不同数据类型行为的硬链接的假设， 排序操作符的定义需要命名一个B-tree操作符族，声明结果数据类型的排序次序。 就像前一节中阐明的，B-tree操作符族定义PostgreSQL的排序概念， 所以这是一个自然的表示。因为point `<->` 操作符返回 `float8` ， 可以在一个操作符类的创建命令中指定，像这样：

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

这里的 `float_ops` 是包含 `float8` 操作的内建操作符族。 这个说明声明了索引可以以 `<->` 操作符的增值的顺序返回行。

35.14.8. 操作符类的特殊特性

还有两种操作符类的特殊特性没有讨论，主要是因为它们对于大多数常用的索引方法并不非常有用。

通常，把一个操作符声明为一个操作符类（或族） 的成员意味着索引方法可以使用该操作符检索满足 `WHERE` 条件的行集合。比如：

```
SELECT * FROM table WHERE integer_column < 4;
```

可以由一个建立在整数字段上的 B-tree 索引精确地满足。但是有时候会有这样的现像： 索引是用作匹配数据行的并不精确的指向。比如，如果一个 GiST 索引只为几何对象存储周界的方块， 那么它就无法精确地满足两个非方形对象(比如多边形)之间是否覆盖的 `WHERE` 条件测试。 但是可以使用这个索引找出那些周界方块和目标对象的周界方块重合的对象， 然后只在索引找到的对象上做精确的重合测试。如果这种情形可以通过，那就说索引对操作符是"松散的"， 松散索引搜索通过当一个行可能或可能不真正满足查询条件时使索引方法返回一个*recheck* 标识来实施。核心系统将然后在检索的行上测试原始的查询条件，以查看是否应该作为一个合法的匹配返回。 如果索引保证返回所有要求的行加上一些附加的行，那么这种方法就可行， 这些额外的行就可以通过执行最初的操作符调用消除。支持松散搜索的索引方法（当前是GiST, SP-GiST 和 GIN） 允许个别的操作符类的支持函数设置recheck标识，所以这是本质上的一个操作符类特征。

再考虑只在索引中存储复杂对象(比如多边形)的周界方块的情形。 这种情况下在索引条目里存储整个多边形没有太多的数值(也可以只存储更简单的 `box` 类型对象)。 这种情形由 `CREATE OPERATOR CLASS` 里的 `STORAGE` 选项存储。可以写类似这样的东西：

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

目前，只有 GiST 和 GIN 索引方法支持与字段数据类型不同的 `STORAGE` 类型。 GiST `compress` 和 `decompress` 支持过程在使用 `STORAGE` 的时候必须处理数据类型转换。对于 GIN 来说， `STORAGE` 类型标识了"键"值的类型，它通常与索引字段的类型不同。比如， 一个用于整数数组字段的操作符类可能正好有整数类型的键。GIN `extractValue` 和 `extractQuery` 支持过程负责从已索引的值抽取键字。

35.15. 包装相关对象到一个扩展

PostgreSQL的一个有用扩展通常包括 多个SQL对象；例如，一个新的数据类型将需要新的函数，新的操作符，以及可能的新的索引操作类。 为了简化数据库管理有助于收集所有这些对象到一个单一的包。 PostgreSQL调用这样的包如*extension*。为了定义一个扩展， 你至少需要一个脚本文件包含SQL命令以创建扩展的对象，并且 控制文件指定扩展本身的几个基本性质。如果扩展包括C代码， 通常也是一个共享库文件的已经编译的C代码。 一旦你有了这些文件，一个简单的CREATE EXTENSION命令加载对象到你的数据库。

使用一个扩展的主要优点，不是运行 SQL脚本加载一组"loose"对象到 你的数据库，而是PostgreSQL将一起了解扩展的对象。 你可以删除使用单独DROP EXTENSION命令的 所有的对象（不需要维护一个单独的"卸载"脚本）。 更有用的，pg_dump知道 它不应该转储扩展的单个成员对象—； 它会只包括 `CREATE EXTENSION` 命令创建扩展。 这大大简化了迁移到一个新的扩展版本， 可能含有比旧版本更多的或不同的对象。 但是请注意，当装载转储到一个新的数据库的时候，你必须有扩展的控制，脚本，以及 其他可用的文件。

PostgreSQL不会让你删除 包含在扩展中的单独的对象，除了减少整个扩展。 同时， 虽然你可以改变扩展成员对象的定义 （例如，通过 `CREATE OR REPLACE FUNCTION` 函数），记住修改后的定义将不会 通过pg_dump被转储。这种变化通常是唯一明智的，如果 你同时在扩展的脚本文件中有相同的变化。 （但对包含配置数据的表有特殊规定；见下文。）

扩展机制也为包装修改脚本制定规定，它调整包含扩展的SQL对象的定义。 例如，如果扩展的版本1.1增加了一个功能并且 改变相比较于1.0的另一个函数的主体，扩展可以提供一个更新脚本，只是那两个改变。 `ALTER EXTENSION UPDATE` 命令可以用于应用这些变化，并且 跟踪扩展的版本，实际上是安装在一个给定的数据库中。

一些SQL对象是显示在ALTER EXTENSION的描述中的扩展对象。 值得注意的是，对象是数据库集群范围，如数据库，角色，和表空间， 无法扩展成员，因为扩展在一个数据库中是唯一已知的。 （虽然并不禁止扩展脚本创建这样的对象，如果它这样做， 他们将不会作为跟踪扩展。）也注意到，当一个表可以是扩展成员时， 其子对象如索引不直接考虑扩展成员。 另外重要的一点是，模式可以属于扩展，但非 反之亦然：扩展这样有一个不合格的名称并且不存在任何模式"内部"。 扩展的成员对象，然而，将属于模式， 只要适合他们的对象类型。它可能或可能不适合拥有模式成员对象的一个扩展。

35.15.1. 扩展文件

CREATE EXTENSION命令依赖于每个扩展的控制文件， 它必须被命名为和带有 `.control` 后缀的扩展相同。并且被放置在安装的 `SHAREDIR/extension` 目录中。必须至少有一个SQL脚本文件， 遵循命名模式 `_extension_` -- `_version_` .sql （比如，扩展 `foo` 的版

本 `1.0` 是 `foo--1.0.sql` ）。 缺省，脚本文件也被放置在 `SHAREDIR/extension` 目录中；但是控制文件可以为脚本文件声明不同的目录。

扩展控制文件的文件格式与 `postgresql.conf` 文件相同， `_parameter_name_` `=` `_value_` 任务列表，每行一个。通过允许 `#` 引进空行和注释。 确保引用任何值，不是单词或者数字。

控制文件可以设置以下参数：

`directory` (`string`)

该目录包含扩展的SQL脚本文件。 除非给定绝对路径名，名字是相对于安装的 `SHAREDIR` 目录。 默认操作相当于指定 `directory = 'extension'` 。

`default_version` (`string`)

扩展的默认版本（如果在 `CREATE EXTENSION` 中没有声明版本，则一个将被安装）。 虽然这可以被省略，如果没有 `VERSION` 选项，这将导致 `CREATE EXTENSION` 失败，所以你通常不想这样做。

`comment` (`string`)

关于扩展的注释（任何字符串）。另外，注释可以通过脚本文件中的COMMENT 命令进行设置。

`encoding` (`string`)

通过脚本文件使用字符集编码。如果脚本文件包含任何非-ASCII字符，则被声明。 否则这些文件被认为数据库编码。

`module_pathname` (`string`)

这个参数的值将 为了每个发生在脚本文件中的 `MODULE_PATHNAME` 被替换。如果它不被设置，则没有替代。通常情况下，这是设置为 `$libdir/``_shared_library_name_` 并且 `MODULE_PATHNAME` 在 `CREATE FUNCTION` 命令中为C语言函数被使用， 因此脚本文件不需要硬线共享库的名字。

`requires` (`string`)

这个扩展取决于扩展名列表，比如 `requires = 'foo, bar'` 。 这些扩展必须在可以被安装前被安装。

`superuser` (`boolean`)

如果这个参数是 `true` （缺省）， 只有超级用户可以创建扩展或者更新它到一个新版本。 如果它被设置为 `false` ， 仅仅需要安装过程中执行命令所需的权限或者更新脚本。

`relocatable` (`boolean`)

如果扩展初始化创建之后可能移动所包含的对象到不同的模式中， 则扩展是浮动的。缺省是 `false` 等，这个扩展是不浮动的。 参见下文获取更多信息。

`schema` (`string`)

这个参数只能设置为非-浮动的扩展。它强制扩展被加载到精确的命名模式中，并且 没有任何其他的。参见下文获取更多信息。

除了初步控制文件 `_extension_` .control， 扩展有在形式 `_extension_` -- `_version_` .control 中命名的二级控制文件。如果被提供，这些必须位于脚本文件目录中。 二级控制文件遵循同样格式作为初步控制文件。 当安装或者更新扩展版本的时候，在二级控制文件中设置的任何参数覆盖初步控制文件， 然而，不能在二级控制文件中设置参数 `directory` 和 `default_version` 。

扩展的SQL脚本文件可以包含任何SQL命令，除了事务控制命令 (`BEGIN` ， `COMMIT` 等)以及不能在一个事务块中执行的命令（比如 `VACUUM` ）。 这是因为脚本文件在事务块中是隐式执行的。

扩展的SQL脚本文件也可以包含以 `\echo` 开头的行， 这被扩展机制忽略（作为注释）。如果脚本文件给psql而不是通过 `CREATE EXTENSION` （参见下文例子脚本）被加载，则这个规定往往抛出错误。 没有那些，用户可能无意中加载扩展内容作为"loose"对象而不是作为扩展， 从中恢复的事态有点繁琐。

当脚本文件可以包含指定编码允许的任何字符时，则控制文件应该包含纯ASCII， 因为PostgreSQL不知道控制文件中的编码方式。 实践中如果你想在扩展注释中使用非-ASCII字符，这个是一个问题。 在这种情况下推荐做法是不使用控制文件 `comment` 参数，但是代替使用 脚本文件中的 `COMMENT ON EXTENSION` 设置comment。

35.15.2. 扩展浮动

用户通常希望加载包含在扩展中的对象到一个扩展者考虑到的不同的模式中。 有浮动的三种支持级别。

- 一个完全可浮动扩展可移动到任何时间下的另一个模式， 即使它被加载到数据库之后。 这是执行了 `ALTER EXTENSION SET SCHEMA` 命令， 它可以自动重命名所有成员对象到新模式中， 通常情况下，这是唯一可能的扩展，如果扩展包含关于任何对象在什么模式中的非内部假设。 同时，扩展的对象都必须在一个模式中（忽略不属于任何模式的对象，如程序语言）。 通过设置控制文件中 `relocatable = true` 标记完全的浮动扩展。
- 一个扩展可能会在安装过程中被重定位，但不是之后。 这是通常的情况，如果扩展的脚本文件需要参考明确的目标模式，例如 在为SQL函数设置 `search_path` 属性时。对于这种扩展， 在控制文件中设置 `relocatable = false` ， 并且使用 `@extschema@` 指向脚本文件中的目标模式。 在执行脚本前该字符串的所有出现将被实际的目标模式的名字取代。 用户可以使用 `CREATE EXTENSION` 的 `SCHEMA` 选项设置目标模式。

- 如果扩展不支持重定位，则在控制文件中设置 `relocatable = false` ，并且设置 `schema` 到目标模式名。 这将防止使用 `CREATE EXTENSION` 的 `SCHEMA` 选项， 除非指定了控制文件命名的相同模式。 如果扩展包含关于模式名不能被 `@extschema@` 替代的内部假设， 这种选择通常是必要的。在这种情况下 `@extschema@` 替代机制可用，尽管 它是有限的使用，因为模式名称是由控制文件确定的。

在所有情况下，脚本文件与search_path初始设置指向 目标模式一起被执行；也就是说， `CREATE EXTENSION` 相当于：

```
SET LOCAL search_path TO @extschema@;
```

这允许通过脚本文件创建的对象到目标模式。如果它希望，则脚本文件可以改变 `search_path` 。 但是这通常是不可取的， `search_path` 被存储到 `CREATE EXTENSION` 的 先前设置完成。

如果它被给定，目标模式是由控制文件中的 `schema` 参数决定的。 否则由 `CREATE EXTENSION` 的 `SCHEMA` 选项决定。 否则当前的默认对象创建模式（调用者 `search_path` 的第一个）。 当使用控制文件 `schema` ，如果它不存在，则创建目标模式，但是在其他两种情况下，它必须已经存在。

如果任何先决条件扩展列在控制文件的 `requires` 中，目标模式附加到 `search_path` 的初始设置中。这允许对象对于新的扩展脚本文件时可见的。

尽管非可重定位扩展可以通过多个模式包含对象，为了外部使用把所有的对象放到一个单独模式中是可取的， 这被认为是扩展的目标模式。在相关扩展创建过程中，这样的安排方便 `search_path` 的缺省设置。

35.15.3. 扩展配置表

一些扩展包含配置表， 其中包含的数据可能安装扩展之后被用户添加或更改。 通常，如果一个表是扩展部分，既不是表的定义， 也不是被pg_dump备份的内容， 但这样的行为对配置表是不需要的；用户修改的任何数据需要包含到备份中， 或备份和重载之后扩展会有不同的表现。

为了解决这个问题，扩展的脚本文件可以标记表，它已经作为配置表被创建，其中 将导致pg_dump包含转储中表的内容（不是定义）。要做到这一点， 在创建表之后调用 `pg_extension_config_dump(regclass, text)` ，比如：

```
CREATE TABLE my_config (key text, value text);

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
```

这种方式可以标记任何数量表。

当 `pg_extension_config_dump` 的第二个参数是空字符串时， 该表的所有内容都被pg_dump备份。如果表 最初扩展脚本创建为空，通常是唯一正确的。如果有 一个初始数据和用户表中提供的数据的混合，则 `pg_extension_config_dump` 的 第二个参数提供了 `WHERE` 条件选择被备份的数据。比如，你可能做

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

并且确保通过扩展脚本创建的行 `standard_entry` 为真。

更加复杂的情况，比如初始化提供的行可能通过用户被修改，通过在配置表上创建触发器被处理 以确保正确标记修改的行。

你可以通过再次调用 `pg_extension_config_dump` 修改与配置表相关的过滤条件。 这在扩展更新脚本中通常是有用的。 标记表不再为配置表的唯一方法是从带有 `ALTER EXTENSION ... DROP TABLE` 的扩展中分离出来。

35.15.4. 扩展更新

扩展机制的一个优点是，它提供了方便管理更新定义一个扩展对象的SQL命令的方式。 这是通过将版本的名称或号链接扩展的安装脚本的每个发布版本做到的。 此外，如果您希望用户可以动态的从一个版本到下一个更新他们的数据库 ，你应该提供*update scripts*执行一个版本到下一个做出必要的改变。 以下模式 `_extension_`--`_oldversion_`--`_newversion_`.sql 更新脚本的名字。（比如， `foo--1.0--1.1.sql` 使用命令修改扩展 `foo` 的版本 `1.0` 到版本 `1.1` ）。

给定一个可用的合适更新脚本， 命令 `ALTER EXTENSION UPDATE` 将更新已安装扩展到指定的新版本。 运行在相同环境中的更新脚本， `CREATE EXTENSION` 提供了安装环境脚本：特别是， `search_path` 以相同 方式进行设置，并通过脚本创建任何新的对象被自动添加到扩展中。

如果扩展有二次控制文件，控制参数用于与脚本目标（新）版本联系的更新脚本。

更新机制可以用来解决一个重要的特殊情况： 将转变"松散"对象的集合到一个扩展。 在扩展机制被添加到PostgreSQL(9.1中)之前， 许多人写的扩展模块简化了已创建的各式各样的未包装的对象。 给定包含这样对象的现有数据库，我们怎么能转换 对象到适当成套扩展？删除它们然后执行纯 `CREATE EXTENSION` 是一种方式， 但它不是可取的，如果对象有依赖关系（例如，如果有扩展创建的数据类型的表列）。 修复这种情况的方式是创建一个空的扩展，然后使用 `ALTER EXTENSION ADD` 把每个预先存在的对象附属在扩展中， 最后在当前扩展版本中创建任何新的对象， 但不在未包装发布中。 `CREATE EXTENSION` 支持带有 `FROM _old_version_` 选项的情况。 这导致它不运行目标版本的正常安装脚本，而是更新脚本命名

`_extension_` -- `_old_version_` -- `_target_version_` .sql。 虚拟版本名称选择使用 `_old_version_` 胜任扩展发起者， 尽管 `未包装` 是一种常见的公约。如果你有多个以前的版本，你需要能够更新扩展风格， 使用多个虚拟版本名称来识别它们。

`ALTER EXTENSION` 能够执行更新 脚本文件序列以实现请求更新。例如，如果只有 `foo--1.0--1.1.sql` 和 `foo--1.1--2.0.sql` 可用， 当目前安装的是 `1.0` 时， 如果需要更新到版本 `2.0` ， `ALTER EXTENSION` 将在序列中应用它们。

PostgreSQL不假定任何有关版本名称的属性： 例如，它不知道 `1.1` 遵循 `1.0` 。 它只匹配可用的版本名称并且 遵循路径要求应用最新的更新脚本。 （一个版本的名称可以是不包含 `--` 或前导或尾随 `-` 的任意字符串。）

有时提供"downgrade"脚本是非常有用的， 例如 `foo--1.1--1.0.sql` 允许恢复与版本 `1.1` 相关的变化。 如果你这样做了，小心downgrade脚本可能会意外地 因它产生一个较短的路径而得以应用的可能性。 风险情况下有一个"快速路径"更新脚本，向前跳几个版本 以及降级脚本到快速路径的起点。 这可能需要较少的步骤应用降级，然后快速路径向前一次移动一个版本。 如果降级脚本删除任何不可替代的对象，这将产生不良的结果。

为了检查意外的更新路径，使用这个命令：

```
SELECT * FROM pg_extension_update_paths('_extension_name_');
```

这显示了已指定扩展的每对不同已知的版本名称， 以及更新路径序列将采取从源版本到目标版本， 或者如果没有可用的更新路径，则为 `NULL` 。 路径以带有 `--` 分隔符的文本形式显示。 如果你喜欢数组形式，则可以使用 `regexp_split_to_array(path,'--')` 。

35.15.5. 扩展实例

这是一个SQL扩展的完整实例，二元复合类型可以存储插槽中的任何类型的值， 被命名为"k"和"v"。非-文本值自动强制转换为文本存储。

脚本文件 `pair--1.0.sql` 看起来像：

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE OR REPLACE FUNCTION pair(anyelement, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(text, anyelement)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(anyelement, anyelement)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = anyelement, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = anyelement, RIGHTARG = text, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = anyelement, RIGHTARG = anyelement, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, PROCEDURE = pair);
```

控制文件 `pair.control` 看起来像：

```
# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
relocatable = true
```

当你几乎不需要makefile安装这两个文件到正确目录时，你可以使用 包含下面内容的 `Makefile` ：

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

这个makefile依赖于PGXS，在Section 35.16中描述。 命令 `make install` 将安装控制和脚本文件到正确目录，通过pg_config 报告。

一旦安装了这些文件，使用CREATE EXTENSION命令加载对象到 任何特定数据库。

35.16. 扩展基础设施建设

如果你正在考虑分配你的PostgreSQL的扩展模块， 为它们设置便携式编译系统相当困难。 因此PostgreSQL安装提供了一个构建 基础设施的扩展，称为PGXS，所以 这个简单的扩展模块可以在已安装的服务器上简单编译。 PGXS的主要目的是为了包含C代码的扩展， 虽然它也可以用于纯SQL扩展。 注意：PGXS不打算 作为一个通用编译系统框架，可用于构建 任何软件接口到PostgreSQL； 它只是为了简单的服务器扩展模块自动化公共建立规则。 对于更复杂的软件包，您可能需要写入自己的构建系统。

为了您的扩展使用PGXS设施， 你必须写一个简单的makefile。 在makefile中，你需要设置一些变量并且 最后包括全局PGXS makefile。 下面是一个例子，建立一个 `isbn_issn` 命名的扩展模块， 由包含一些C代码，扩展的控制文件，SQL脚本，和文本文件的共享库组成：

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

最后三行总是相同的。早在文件中，你可以指定变量或者添加自定义make 规则。

设置这三个变量之一指定建立什么：

`MODULES`

从源文件同一地方编译共享库对象列表（不包括列表中的库后缀）

`MODULE_big`

从多个源文件构建共享库（在 `OBJS` 中列出对象文件）

`PROGRAM`

建立可执行程序（在 `OBJS` 中列出对象文件）

下面的变量也可以设置：

`EXTENSION`

扩展名：对于每一个名字你必须提供一个 `_extension_` .control 文件，它将被安装到 `_prefix_` /share/extension中。

`MODULEDIR`

DATA和DOCS文件应该被安装到 `_prefix_` /share 子目录中（如果没有设置，如果设置 `EXTENSION` ， 缺省是 `extension` 。如果没有则为 `contrib` ）。

`DATA`

随机文件安装到 `_prefix_` /share/$MODULEDIR。

`DATA_built`

随机文件安装到 `_prefix_` /share/$MODULEDIR， 这首先需要编译。

`DATA_TSEARCH`

随机文件安装到 `_prefix_` /share/tsearch_data中。

`DOCS`

随机文件安装到 `_prefix_` /doc/$MODULEDIR

`SCRIPTS`

脚本文件（非二进制数）安装到 `_prefix_` /bin中

`SCRIPTS_built`

脚本文件（非二进制数）安装到 `_prefix_` /bin， 这需要首先编译。

`REGRESS`

回归测试用例列表（没有后缀）， 参见下文。

`REGRESS_OPTS`

另外切换到pg_regress

`EXTRA_CLEAN`

`make clean` 中删除额外文件

`PG_CPPFLAGS`

被添加到 `CPPFLAGS`

`PG_LIBS`

被添加到 `PROGRAM` 连接线

`SHLIB_LINK`

被添加到 `MODULE_big` 连接线

`PG_CONFIG`

为了PostgreSQL安装编译其路径指向pg_config 应用程序（通常 `pg_config` 使用你的 `PATH` 中的第一个）。

把这个makefile作为 `Makefile` 放在持有你的扩展的目录中。 然后你可以执行 `make` 编译， 然后 `make install` 安装模块。默认情况下，为PostgreSQL对应于 你的 `PATH` 中找到的第一个 `pg_config` 程序扩展被 编译安装。你可以通过设置 `PG_CONFIG` 指向 `pg_config` 程序来使用

一个不同的安装， 或者在makefile中或在 `make` 命令行上。

Caution

当编译PostgreSQL 8.3或更高版本时， 改变 `PG_CONFIG` 。老版本不工作设置它除了 `pg_config` ；你必须改变你的 `PATH` 来选择编译安装。

在 `REGRESS` 变量中列出的脚本用于 你的模块的回归测试，这可以在执行 `make install` 之后通过 `make installcheck` 调用。 为了可以运行你必须有一个运行的PostgreSQL服务器。
在 `REGRESS` 中的脚本文件必须出现在您的扩展目录中的 `sql/` 命名的子目录中。这些文件必须有扩展 `.sql` ， 这没有包含在makefile列出的 `REGRESS` 中。 每个测试还应在 `expected/` 命名的子目录中包含一个预期的输出文件， 以及相同的词干和扩展 `.out` 。
`make installcheck` 执行每个psql的测试脚本， 并且比较结果输出到匹配期望的文件。 任何差异会写入 `diff -c` 格式的文件 `regression.diffs` 中。 请注意，试图运行一个测试，缺少预期的文件将被作为"问题"报告， 所以确保你有所有预期的文件。

> **Tip:** 创造期望文件的最简单的方法是创建空文件， 然后做一个测试运行（这当然会报告差异）。检查 在 `results/` 目录中发现的实际结果文件，如果匹配从这个试验中你期望的， 那么将它们复制到 `expected/` 。

Chapter 36. 触发器

Table of Contents

本章提供有关书写触发器函数的一般信息。 触发器函数可以用大多数过程语言书写， 包括 PL/pgSQL (Chapter 40), PL/Tcl (Chapter 41), PL/Perl (Chapter 42)和 PL/Python (Chapter 43)。 阅读完本章之后，你应该参考你喜欢的过程语言的章节， 找出使用这些语言书写触发器的一些语言相关的细节。

也可以用C来写触发器，不过大多数人都会觉得使用某种过程语言书写更简单。 目前还不能用简单的 SQL 函数语言书写触发器函数。

36.1. 触发器行为概述

一个触发器是一种声明，告诉数据库应该在执行特定的操作的时候执行特定的函数。 触发器可以附加到表和视图上。

触发器可以定义在一个 `INSERT` ， `UPDATE` , or `DELETE` 命令之前或者之后执行， 要么是对每行执行一次，要么是对每条SQL语句执行一次。 如果某列在 `UPDATE` 语句的 `SET` 子句中被提及，则 `UPDATE` 触发器再次被触发。 触发器可以为 `TRUNCATE` 语句触发。 如果发生触发器事件，那么将在合适的时刻调用触发器函数以处理该事件。

在视图上，触发器可以被定义执行而不是 `INSERT` ， `UPDATE` 或者 `DELETE` 操作。为了需要在视图中修改的每一行触发 `INSTEAD OF` 触发器。 这是触发器函数在基表下执行必要修改的责任，并且在适当情况下，返回在视图中出现的修改的行。 在执行每个SQL语句， `INSERT` ， `UPDATE` 或者 `DELETE` 操作之前或之后也可以定义视图上的触发器。

触发器函数必须在创建触发器之前，作为一个没有参数并且返回 `trigger` 类型的函数定义。 触发器函数通过特殊的 `TriggerData` 结构接收其输入， 而不是用普通的函数参数方式。

一旦创建了一个合适的触发器函数， 就可以用CREATE TRIGGER创建触发器。 同一个触发器函数可以用于多个触发器。

PostgreSQL提供按行与按语句触发的触发器。 按行触发的触发器函数为触发语句影响的每一行执行一次；相比之下， 按语句触发的触发器函数为每条触发语句执行一次，而不管影响的行数。 特别是，一个影响零行的语句将仍然导致按语句触发的触发器执行。 这两种类型的触发器有时候分别叫做行级触发器和语句级触发器。 在 `TRUNCATE` 上的触发器可能只能在语句级别定义。 触发之前或之后的视图，触发器只能在语句级别定义， 然而非 `INSERT` ， `UPDATE` 或者 `DELETE` 触发的触发器在行级别定义。

触发器通常按照触发的*before*和*after*， 或者*instead of*操作进行分类。 这些分别被称为 `BEFORE` 触发器， `AFTER` 触发器， `INSTEAD OF` 触发器。 语句级别的 `BEFORE` 触发器通常在语句开始做任何事情之前触发， 而语句级别的 `AFTER` 触发器在语句结束时触发。 触发器的这些类型可以在表或者视图上定义。 行级别的 `BEFORE` 触发器在对特定行进行操作之前触发，而行级别的 `AFTER` 触发器在语句结束的时候触发 (但是在任何语句级别的 `AFTER` 触发器之前)。 触发器的这些类型可能只在表上定义。 行级别 `INSTEAD OF` 触发器可能只在视图上定义， 并且立刻触发作为视图上的每一行被标识为需要的操作。

按语句触发的触发器应该总是返回 `NULL` 。 如果必要，按行触发的触发器函数可以给调用它的执行者返回一行数据 (一个类型为 `HeapTuple` 的数值)， 那些在操作之前触发的触发器有以下选择：

- 它可以返回 `NULL` 以忽略对当前行的操作。 这就指示执行器不要执行调用该触发器的行级别操作(对特定行的插入或者更改)。

- 只用于 `INSERT` 和 `UPDATE` 行触发器：返回的行将成为被插入的行或者是成为将要更新的行。这样就允许触发器函数修改将要被插入或者更新的行。

一个无意导致任何这类行为的在操作之前触发的行级触发器必须仔细返回那个被当作新行传进来的行。也就是说，对于 `INSERT` 和 `UPDATE` 触发器而言，是 `NEW` 行，对于 `DELETE` 触发器而言，是 `OLD` 行。

行级别 `INSTEAD OF` 触发器应该返回 `NULL` 表示它不修改来自视图的基础表的任何数据，它应该返回传递到（ `INSERT` 和 `UPDATE` 的 `NEW` 行，或者 `DELETE` 操作的 `OLD` 行）的视图行。非空返回值用于发信号，使触发器执行视图中必要的数据修改。这将导致计算通过这个命令递增的受影响的行数。对于 `INSERT` 和 `UPDATE` 操作，触发器可能在返回它之前修改 `NEW` 行。这将改变通过 `INSERT RETURNING` 或者 `UPDATE RETURNING` 返回的数据。并且当视图不能完全显示所提供的同一数据时是有用的。

对于在操作之后触发的行级触发器，其返回值会被忽略，因此可以返回 `NULL` 。

如果多于一个触发器为同样的事件定义在同样的关系上，触发器将按照名字的字母顺序触发。在 `BEFORE` 和 `INSTEAD OF` 触发器的情况下，每个触发器返回的可能已经被修改过的行成为下一个触发器的输入。如果 `BEFORE` 或者 `INSTEAD OF` 触发器返回 `NULL` ，那么对该行的操作将被丢弃并且随后的触发器也不会被触发。

一个触发器定义也可以声明一个布尔型的 `WHEN` 条件，用于检查触发器是否应该被触发。在行级别触发器上，`WHEN` 条件可以检查旧和/或新的列值。语句级的触发器也可以有 `WHEN` 条件，尽管对其没有用。在一个 `BEFORE` 触发器中，`WHEN` 条件只在函数正在或将被执行之前被触发执行，因此使用 `WHEN` 条件实际上与在触发器开始时执行相同条件的结果是一样的。然而，在一个 `AFTER` 触发器中，`WHEN` 条件只有在发生更新行时才会执行，并且决定在语句结束之后，一个事件是否需要等待触发触发器。因此当一个 `AFTER` 触发器的 `WHEN` 条件没有返回真时，队列中的时间不需要在语句结束后重新读取行。如果触发器只会被一些行触发时，`INSTEAD OF` 触发器不支持 `WHEN` 条件。

通常，行的 `BEFORE` 触发器用于检查或修改将要插入或者更新的数据。比如，一个 `BEFORE` 触发器可以用于把当前时间插入一个 `timestamp` 字段，或者跟踪该行的两个元素是一致的。行的 `AFTER` 触发器多数用于填充或者更新其它表，或者对其它表进行一致性检查。这么区分工作的原因是 `AFTER` 触发器肯定可以看到该行的最后数值，而 `BEFORE` 触发器不能；还可能有其它的 `BEFORE` 触发器在其后触发。如果你没有具体的原因定义触发器是 `BEFORE` 或者 `AFTER` ，那么 `BEFORE` 触发器的效率高些，因为操作相关的信息不必保存到语句的结尾。

如果一个触发器函数执行SQL命令，而这些命令再次触发触发器，这就是所谓的级联触发器。对级联触发器的级联深度没有明确的限制。有可能出现级联触发器导致同一个触发器递归调用的情况；比如，一个 `INSERT` 触发器可能执行一个命令，把一个额外的行插入同一个表中，导致 `INSERT` 触发器再次触发。避免这样无穷递归的问题是触发器程序员的责任。

在定义一个触发器的时候，可以声明一些参数。 在触发器定义中包含参数的目的是允许类似需求的不同触发器调用同一个函数。 比如，可能有一个通用的触发器函数，接受两个字段名字，把当前用户放在第一个， 而当前时间戳在第二个。只要写得恰当， 那么这个触发器函数就可以和触发它的特定表无关。 这样同一个函数就可以用于有着合适字段的任何表的 `INSERT` 事件， 实现自动跟踪交易表中的记录创建之类的问题。 如果定义成一个 `UPDATE` 触发器，还可以用它跟踪最后更新的事件。

每种支持触发器的编程语言都有自己的方法让触发器函数得到输入数据。 这些输入数据包括触发器事件的类型(比如 `INSERT` 或者 `UPDATE`)以及所有 在 `CREATE TRIGGER` 里面列出的参数。 对于低层次的触发器， 输入数据也包括 `INSERT` 和 `UPDATE` 触发器的 `NEW` 和/或 `UPDATE` 和 `DELETE` 触发器的 `OLD` 行。 语句级别的触发器目前没有任何方法检查该语句修改的独立行。

36.2. 数据改变的可视性

如果在触发器函数里执行SQL命令，并且这些命令访问触发器所在的表， 那么你必须知道触发器的可视性规则， 因为这些规则决定这些SQL命令是否能看到触发触发器的数据改变。 简单说：

- 语句级别的触发器遵循简单的可视性原则： 在语句之前(before)触发的触发器看不到语句所做的修改， 而所有修改都可以被 `AFTER` 语句级别触发的触发器看到。
- 导致触发器触发的数据改变(插入、更新、 删除)通常是不能被 一个 `BEFORE` 触发器里面执行的SQL命令看到的，因为它还没有发生。
- 不过，在 `BEFORE` 触发器里执行的SQL 命令将能够看到在同一个外层命令前面处理的行的改变。 这一点需要仔细，因为这些改变的顺序通常是不可预期的； 一个影响多行的SQL命令可能以任意顺序访问这些行。
- 同样的，行级别 `INSTEAD OF` 触发器将看到通过先前同一外部命令 触发的 `INSTEAD OF` 触发器形成的数据变化的影响。
- 在一个行级 `AFTER` 触发器被触发的时候， 所有外层命令产生的数据改变都已经完成， 对于触发的触发器函数是可见的。

如果是用任何一种标准过程语言写的触发器函数， 那么只有当函数声明了 `VOLATILE` 才会应用上面的语句。 声明了 `STABLE` 或者 `IMMUTABLE` 的函数在任何情况 下都不会看到请求中做出的改变。

有关数据可视性规则的更多信息可以在Section 44.4找到。 Section 36.4里的例子包含这些规则的演示。

36.3. 用C写触发器

本章描述触发器函数的低层细节。只有当你用C书写触发器函数的时候才需要这些信息。 如果你用某种高级语言写触发器，那么系统就会为你处理这些细节。 在大多数情况下，你在书写自己的C触发器之前应该考虑使用过程语言。 每种过程语言的文档里面都有关于如何用该语言书写触发器的解释。

触发器函数必须使用"version 1"的函数管理器接口。

当一个函数被触发器管理器调用时，它不会收到任何普通参数， 而是收到一个指向 `TriggerData` 结构的"context"指针。 C函数可以通过执行下面的宏来检查它们是否是从触发器管理器调用的

```
CALLED_AS_TRIGGER(fcinfo)
```

扩展到：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

如果此宏返回真(TRUE)，则可以安全地把 `fcinfo->context` 转换成 `TriggerData*` 类型，然后使用这个指向 `TriggerData` 的结构。 函数本身绝不能更改 `TriggerData` 结构或者它指向的任何数据。

`struct TriggerData` 是在 `commands/trigger.h` 中被定义的：

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

这些成员的定义如下：

`type`

总是 `T_TriggerData` 。

`tg_event`

描述调用函数的事件。你可以用下面的宏检查 `tg_event` ：

`TRIGGER_FIRED_BEFORE(tg_event)`

如果触发器是在操作前触发，返回真。

`TRIGGER_FIRED_AFTER(tg_event)`

如果触发器是在操作后触发，返回真。

`TRIGGER_FIRED_INSTEAD(tg_event)`

如果触发器触发了相反的操作，返回真。

`TRIGGER_FIRED_FOR_ROW(tg_event)`

如果触发器是行级别事件触发，返回真。

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

如果触发器是语句级别事件触发，返回真。

`TRIGGER_FIRED_BY_INSERT(tg_event)`

如果触发器是由 `INSERT` 触发，返回真。

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

如果触发器是由 `UPDATE` 触发，返回真。

`TRIGGER_FIRED_BY_DELETE(tg_event)`

如果触发器是由 `DELETE` 触发，返回真。

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

如果触发器是由 `TRUNCATE` 命令触发，返回真。

`tg_relation`

是一个指向描述被触发的关系的结构的指针。 请参考 `utils/rel.h` 获取关于此结构的详细信息。 最让人感兴趣的事情是 tg_relation->rd_att (关系行的描述) 和 tg_relation->rd_rel->relname (关系名。这个变量的类型不是 `char*` 而是 `NameData` 。 如果你需要一份名字的拷贝， 用 `SPI_getrelname(tg_relation)` 获取 `char*`)。

`tg_trigtuple`

是一个指向触发触发器的行的指针。 这是一个正在被插入(INSERT)、删除(DELETE)、或更新(UPDATE)的行。 如果是 `INSERT` 或者 `DELETE` ， 如果你不想用另一条行覆盖此行（就 `INSERT` 来说）或忽略操作， 那么这就是你将从函数返回的东西。

`tg_newtuple`

如果是 `UPDATE` ， 这是一个指向新版本的行的指针， 如果是 `INSERT` 或者 `DELETE` ， 则是 `NULL` 。 如果事件是 `UPDATE` 并且你不想用另一条行替换这条行或忽略操作的话， 这就是你将从函数返回的东西。

`tg_trigger`

是一个指向结构 `Trigger` 的指针，该结构在 `utils/reltrigger.h` 里定义：

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
} Trigger;
```

`tgname` 是触发器的名称， `tgnargs` 是在 `tgargs` 里参数的数量， `tgargs` 是一个指针数组，数组里每个指针指向在 `CREATE TRIGGER` 语句里声明的参数。 其它成员只在内部使用。

`tg_trigtuplebuf`

如果没有这样的元组或者没有存储在磁盘缓冲区里， 则是包含 `tg_trigtuple` 或者 `InvalidBuffer` 的缓冲区。

`tg_newtuplebuf`

如果没有这样的元组或者它并未存储在磁盘缓冲区里， 那么就是包含 `tg_newtuple` 或者 `InvalidBuffer` 的缓冲区。

一个触发器函数必须返回一个 `HeapTuple` 指针或者一个 `NULL` 指针(_不是_SQL的NULL值，也就是说不要设置 `isNull` 为真)。 请注意如果你不想修改正在被操作的行，那么要根据情况返回 `tg_trigtuple` 或者 `tg_newtuple` 。

36.4. 一个完整的触发器例子

这里是一个用C写的非常简单的触发器例子。（用程序语言写的触发器例子可以在程序语言文档中找到）。

函数 `trigf` 报告 `ttest` 表的行数量， 并且如果命令试图把NULL插入到字段 `x` 里 (也就是它做为一个非空约束但不退出事务)时略过操作。

首先，表定义：

```
CREATE TABLE ttest (
    x integer
);
```

这里是触发器函数的源代码：

```
#include "postgres.h"
#include "executor/spi.h"        /* 你用 SPI 的时候要用的头文件 */
#include "commands/trigger.h"   /* ... 触发器 ... */
#include "utils/rel.h"          /* ... 和关系 */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* 确信自己是作为触发器调用的 */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* 返回给执行者的行 */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* 检查 NULL 值 */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";
```

```
    tupdesc = trigdata->tg_relation->rd_att;

    /* 与 SPI 管理器连接 */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* 获取表中的行数量 */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) 返回 int8 , 所以要小心转换  */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

编译完源代码后(参见Section 35.9.6)，声明函数并创建触发器：

```
CREATE FUNCTION trigf() RETURNS trigger
    AS '_filename_'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

现在你可以测试触发器的操作：

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- 插入被忽略, AFTER 触发器没有触发

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                             还记得讲过的关于可视性的原则吗？
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                             还记得讲过的关于可视性的原则吗？
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                             还记得讲过的关于可视性的原则吗？
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

在 `src/test/regress/regress.c` 和 spi里还有更复杂的例子。

Chapter 37. 事件触发器

Table of Contents

为了补充在Chapter 36讨论过的触发器机制，PostgreSQL 也提供了事件触发器。不同于规则触发器附加到一个表上并且只捕获DML事件， 事件触发器对一个特定的数据库是全局的，并且可以捕获DDL事件。

像规则触发器一样，事件触发器可以用任何包括事件触发器支持的过程语言来写， 或者用C，但是不能用简单的SQL。

37.1. 事件触发器行为的概述

当事件以它定义的方式在数据库中相关的出现时，事件触发器触发。当前支持的事件是 `ddl_command_start`，`ddl_command_end` 和 `sql_drop`。 额外事件的支持将在将来的版本中添加。

`ddl_command_start` 事件只在 `CREATE`，`ALTER`，或 `DROP` 命令执行之前发生。在事件触发器触发之前并不检查受影响的对象是否存在。然而，作为一个例外， DDL命令针对共享对象时这个事件并不触发 — 数据库，角色和表空间 — 或命令针对事件触发器它们自己时也不触发。事件触发器机制不支持这些对象类型。 `ddl_command_start` 也在 `SELECT INTO` 命令执行之前触发，因为这相当于 `CREATE TABLE AS`。

`ddl_command_end` 事件只在相同的命令集的执行之后触发。

`sql_drop` 事件只在删除数据库对象的任意操作触发 `ddl_command_end` 事件之前发生。 要列出被删除的对象，使用 `sql_drop` 事件触发器代码的设置返回函数 `pg_event_trigger_dropped_objects()` （参阅Section 9.28）。 请注意，触发器在对象已经从系统目录中删除以后执行，所以不可能在看到他们了。

事件触发器（类似其他函数）不能在一个中止的事务中执行。因此， 如果一个DDL命令错误失败，相关的 `ddl_command_end` 触发器将不会执行。 相反的，如果一个 `ddl_command_start` 触发器错误失败了，将不会有更多的触发器触发， 也不会尝试执行命令本身。相似的，如果一个 `ddl_command_end` 触发器错误失败了， DDL语句的影响将会回滚，就像他们在任何其他包含事务中止的情况下一样。

事件触发器机制支持的命令的完整列表，请参阅Section 37.2。

使用命令CREATE EVENT TRIGGER创建事件触发器。为了创建事件触发器， 必须首先创建一个特殊返回类型为 `event_trigger` 的函数。 这个函数不需要（或不可能）返回一个值；返回类型只是作为一个该函数被作为一个事件触发器调用的信号服务的。

如果为一个特别事件定义了多个事件触发器，他们将按照触发器名字的字母顺序触发。

触发器定义也可以声明一个 `WHEN` 条件，例如，一个 `ddl_command_start` 触发器可以只为用户想要拦截的特别的命令触发。这样的触发器共同使用的是限制用户可能执行的DDL操作的范围。

37.2. 事件触发器触发矩阵

Table 37-1列出了所有支持事件触发器的命令。

Table 37-1. 命令标记支持的事件触发器

命令标记	ddl_command_start	ddl_command_end	sql_drc
ALTER AGGREGATE	X	X	-
ALTER COLLATION	X	X	-
ALTER CONVERSION	X	X	-
ALTER DOMAIN	X	X	-
ALTER EXTENSION	X	X	-
ALTER FOREIGN DATA WRAPPER	X	X	-
ALTER FOREIGN TABLE	X	X	X
ALTER FUNCTION	X	X	-
ALTER LANGUAGE	X	X	-
ALTER OPERATOR	X	X	-
ALTER OPERATOR CLASS	X	X	-
ALTER OPERATOR FAMILY	X	X	-
ALTER SCHEMA	X	X	-
ALTER SEQUENCE	X	X	-
ALTER SERVER	X	X	-
ALTER TABLE	X	X	X
ALTER TEXT SEARCH CONFIGURATION	X	X	-
ALTER TEXT SEARCH DICTIONARY	X	X	-
ALTER TEXT SEARCH PARSER	X	X	-
ALTER TEXT SEARCH TEMPLATE	X	X	-
ALTER TRIGGER	X	X	-
ALTER TYPE	X	X	-
ALTER USER MAPPING	X	X	-
ALTER VIEW	X	X	-
CREATE AGGREGATE	X	X	-
CREATE CAST	X	X	-
CREATE COLLATION	X	X	-
CREATE CONVERSION	X	X	-
CREATE DOMAIN	X	X	-
CREATE EXTENSION	X	X	-

`CREATE FOREIGN DATA WRAPPER`	X	X	-
`CREATE FOREIGN TABLE`	X	X	-
`CREATE FUNCTION`	X	X	-
`CREATE INDEX`	X	X	-
`CREATE LANGUAGE`	X	X	-
`CREATE OPERATOR`	X	X	-
`CREATE OPERATOR CLASS`	X	X	-
`CREATE OPERATOR FAMILY`	X	X	-
`CREATE RULE`	X	X	-
`CREATE SCHEMA`	X	X	-
`CREATE SEQUENCE`	X	X	-
`CREATE SERVER`	X	X	-
`CREATE TABLE`	X	X	-
`CREATE TABLE AS`	X	X	-
`CREATE TEXT SEARCH CONFIGURATION`	X	X	-
`CREATE TEXT SEARCH DICTIONARY`	X	X	-
`CREATE TEXT SEARCH PARSER`	X	X	-
`CREATE TEXT SEARCH TEMPLATE`	X	X	-
`CREATE TRIGGER`	X	X	-
`CREATE TYPE`	X	X	-
`CREATE USER MAPPING`	X	X	-
`CREATE VIEW`	X	X	-
`DROP AGGREGATE`	X	X	X
`DROP CAST`	X	X	X
`DROP COLLATION`	X	X	X
`DROP CONVERSION`	X	X	X
`DROP DOMAIN`	X	X	X
`DROP EXTENSION`	X	X	X
`DROP FOREIGN DATA WRAPPER`	X	X	X
`DROP FOREIGN TABLE`	X	X	X
`DROP FUNCTION`	X	X	X
`DROP INDEX`	X	X	X
`DROP LANGUAGE`	X	X	X
`DROP OPERATOR`	X	X	X
`DROP OPERATOR CLASS`	X	X	X
`DROP OPERATOR FAMILY`	X	X	X
`DROP OWNED`	X	X	X

`DROP RULE`	X	X	X
`DROP SCHEMA`	X	X	X
`DROP SEQUENCE`	X	X	X
`DROP SERVER`	X	X	X
`DROP TABLE`	X	X	X
`DROP TEXT SEARCH CONFIGURATION`	X	X	X
`DROP TEXT SEARCH DICTIONARY`	X	X	X
`DROP TEXT SEARCH PARSER`	X	X	X
`DROP TEXT SEARCH TEMPLATE`	X	X	X
`DROP TRIGGER`	X	X	X
`DROP TYPE`	X	X	X
`DROP USER MAPPING`	X	X	X
`DROP VIEW`	X	X	X
`SELECT INTO`	X	X	-

37.3. 用C编写事件触发器函数

本节描述事件触发器函数接口的底层细节。只有在用C编写事件触发器函数的时候才会需要这些信息。 如果你使用一种高级语言，那么这些细节已经为你处理了。在大多数情况下， 应该在用C编写事件触发器之前考虑用过程语言。每一个过程语言的文档解释了如何用那种语言编写一个事件触发器。

事件触发器函数必须使用"version 1"函数管理接口。

当通过事件触发器管理调用一个函数的时候，并不传递任何正规参数，而是传递一个"context"指针指向 `EventTriggerData` 结构。 C函数可以检查它们是从事件触发器管理调用还是通过执行宏：

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

扩展到：

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

如果返回真，那么传递 `fcinfo->context` 到类型 `EventTriggerData *` 是安全的，并且利用指向的 `EventTriggerData` 结构。 函数必须不改变 `EventTriggerData` 结构或它指向的任何数据。

`struct EventTriggerData` 是在 `commands/event_trigger.h` 里定义的：

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* event name */
    Node       *parsetree;  /* parse tree */
    const char *tag;        /* command tag */
} EventTriggerData;
```

成员定义如下：

`type`

总是 `T_EventTriggerData` 。

`event`

描述了函数调用的事件，是 `"ddl_command_start"` ， `"ddl_command_end"` ， `"sql_drop"` 之一。参阅Section 37.1获得这些事件的含义。

`parsetree`

命令的分析树的一个指针。检查PostgreSQL源代码详情。分析树结构改变时不会有通知。

`tag`

命令标签和事件触发器正在运行的事件有关，例如 `"CREATE FUNCTION"` 。

一个事件触发器函数必须返回一个 `NULL` 指针（不是 SQL null值，也就是，不要设置 `isNull` 为真）。

37.4. 一个完整的事件触发器的例子

这是用C编写的事件触发器函数的一个非常简单的例子。（用过程语言编写的触发器的例子可以在过程语言的文档中找到。）

函数 `noddl` 在每次调用它时提出一个异常。事件触发器定义在 `ddl_command_start` 事件时联系函数。影响是阻止所有DDL命令（除了在Section 37.1中提到的例外）运行。

这是触发器函数的源代码：

```
#include "postgres.h"
#include "commands/event_trigger.h"

PG_MODULE_MAGIC;

Datum noddl(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
        (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

在编译源代码之后（参阅Section 35.9.6），声明函数和触发器：

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE PROCEDURE noddl();
```

现在可以测试触发的操作：

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Procedure | Tags
-------+-------------------+-------+---------+-----------+------
 noddl | ddl_command_start | dim   | enabled | noddl     |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

在这种情况下，当你需要做这些的时候为了能够运行一些DDL命令，你必须终止事件触发器或禁用它。 只在事务期间禁用触发器是方便的：

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

（回想一下，事件触发器自己的DDL命令不受事件触发器的影响。）

Chapter 38. 规则系统

Table of Contents

本章讨论PostgreSQL里的规则系统。 生产规则系统的概念是很简单的，但是在实际使用的时候会碰到很多细节问题。

有些其它数据库系统定义动态的数据库规则。这些通常是存储过程和触发器， 在PostgreSQL里，这些东西也可以通过函数和触发器来实现。

规则系统(更准确地说是查询重写规则系统)是和存储过程和触发器完全不同的东西。 它把查询修改为需要考虑规则的形式，然后把修改过的查询传递给查询规划器执行。 这是非常有效的工具并且可以用于许多像查询语言过程、视图、版本等。 这个规则系统的理论基础和能力在 数据库系统中的规则，过程，缓存和视图 和数据库系统中为版本模型使用生产规则的统一框架里有讨论。

38.1. 查询树

要理解规则系统如何工作，首先要知道规则何时被激发以及它的输入和结果是什么。

规则系统位于分析器和规划器之间。以分析器输出的查询树以及用户定义的重写规则作为输入， 重写规则也是一个查询树，只不过增加了一些扩展信息，然后创建零个或者多个查询树作为结果。 所以它的输入和输出仍然是那些分析器可以生成的东西，因而任何规则系统看到的东西都是可以用 SQL语句表达的。

那么什么是查询树呢?它是一个SQL语句的内部表现形式， 这时组成该语句的每个独立部分都是分别存储的。如果你设置了配置参数 `debug_print_parse`，`debug_print_rewritten`，或 `debug_print_plan`，那么就可以在服务器日志中看到这些查询树。 规则动作也是以查询树的方式存储的，存放在系统表 `pg_rewrite` 里面。 不过不是用像调试输出那样的格式，但包含的内容是完全一样的。

阅读一个裸查询树需要一定的经验，但是因为理解查询树的SQL 表现就足以理解规则系统，所以这份文档将不会告诉你如何读取它们。

当读取本章中查询树的SQL表现时， 必须能够识别该语句被分解后放在查询树里的成员。查询树的成员有

命令类型(command type)

这是一个简单的值，说明哪条命令(`SELECT`，`INSERT`，`UPDATE`，`DELETE`)生成这个查询树。

范围表(range table)

范围表是一个查询中使用的关系的列表。在 `SELECT` 语句里就是在 `FROM` 关键字后面给出的关系。

每个范围表表示一个表或一个视图，表明是查询里哪个成员调用了它。在查询树里， 范围表是用代号而不是用名字引用的，所以这里不用像在SQL 语句里一样关心是否有重名问题。这种情况在引入了规则的范围表后可能会发生。 本章的例子将不讨论这种情况。

结果关系(result relation)

这是一个范围表的索引，用于标识查询结果前往的表。

`SELECT` 查询通常没有结果关系。特例 `SELECT INTO` 几乎等于一个跟随 `INSERT ... SELECT` 的 `CREATE TABLE`，所以这里就不单独讨论了。

在 `INSERT`，`UPDATE`，`DELETE` 命令里， 结果关系是更改发生影响的表或视图。

目标列(target list)

目标列是一列定义查询结果的表达式。在 `SELECT` 的情况下， 这些表达式就是构建查询的最终输出的东西。它们是位于 `SELECT` 和 `FROM` 关键字之间的表达式(`*` 只是表明一个关系的所有字段的缩写，它被分析器扩展为独立的字段，因此规则系统永远看不到它)。

`DELETE` 不需要正常的目标列是因为它们不产生任何结果。 相反的，规划器会向空目的标列中增加一条特殊的CTID记录， 以允许执行器找到被删除的行。（当结果关系是一个普通的表时添加CTID。 如果结果关系是一个视图，则添加所有的行变量，描述在Section 38.2.4。）

对于 `INSERT` 命令里面，目标列描述了应该进入结果关系的新行。 这些行由那些在 `VALUES` 子句里的表达式或在 `INSERT ... SELECT` 语句里的 `SELECT` 子句里面的表达式构成。 重写过程的第一步就是为任何不是由原始的查询赋值，并且有缺省值的字段增加目标列表项。 任何其它的字段(既无给出值也无缺省值)将由规划器自动赋予一个常量 NULL 表达式。

对于 `UPDATE` 命令，目标列描述应该替换旧行的新行。在规则系统里， 它只包含来自命令的 `SET column = expression` 部分抽取的表达式。 这时，规划器将通过插入从旧行抽取数据到新行的表达式的方法处理缺失的字段。 就像对 `DELETE` ，规则系统添加CTID或整行变量使执行者可以识别要被更新的旧行。

目标列里的每个元素都包含着一个表达式，它可以为常量值、一个指向某个范围表里面的关系的一个字段的变量、 一个参数、一个由函数调用/常量/变量/操作符等构成的表达式树。

条件(qualification)

查询条件是一个表达式，它非常类似那些包含在目标列里的条目。这个表达式的值是一个布尔值， 通过此值来判断对最终结果行是否要执操作(`INSERT` , `UPDATE` , `DELETE` , 或 `SELECT`)。它是一个SQL 语句的 `WHERE` 子句。

连接树(join tree)

查询的连接树显示了 `FROM` 子句的结构。对于像 `SELECT ... FROM a, b, c` 这样的简单查询，连接树只是一个 `FROM` 项的简单列表，因为允许以任意顺序连接它们。 但如果使用了 `JOIN` 表达式(尤其是外连接的时候)，就必须按照该连接显示的顺序进行连接。 在这种情况下，连接树显示 `JOIN` 表达式的结构。与特定的 `JOIN` 子句(来自 `ON` 或 `USING` 表达式)相关的限制做为附加在那些连接树节点的条件表达式存储。 事实证明把顶层 `WHERE` 表达式也当做附加在顶层连接树项的条件来存储是非常方便的。 所以实际上连接树代表 `SELECT` 语句的 `FROM` 和 `WHERE` 子句。

其它(others)

查询树的其它部分，像 `ORDER BY` 子句，不准备在这里讨论。 规则系统在附加规则时将在那里(ORDER BY 子句)替换一些条目，但是这对于规则系统的基本原理并没有多大关系。

38.2. 视图和规则系统

PostgreSQL里的视图是通过规则系统来实现的。下面的命令：

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

实际上和下面两条命令：

```
CREATE TABLE myview (_same column list as mytab_);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

之间本质上没有区别，因为这就是 `CREATE VIEW` 命令在内部实际执行的内容。 这样做有一些负作用。其中之一就是在PostgreSQL 系统表里的视图的信息与一般表的信息完全一样。所以对于查询分析器来说， 表和视图之间完全没有区别。它们是同样的事物：关系。

38.2.1. `SELECT` 规则如何运转

`ON SELECT` 的规则在最后一步应用于所有查询，哪怕给出的是一条 `INSERT`，`UPDATE` 或 `DELETE` 命令。而且与其它命令类型上的规则有不同的语意， 那就是它们在现场修改查询树而不是创建一个新的查询树。所以先介绍 `SELECT` 规则。

目前，一个 `ON SELECT` 规则里只能有一个动作，而且它必须是一个无条件的 `INSTEAD` (取代) 的 `SELECT` 动作。有这个限制是为了令规则安全到普通用户也可以打开它们， 并且它限制 `ON SELECT` 规则使之行为类似视图。

本文档的例子是两个连接视图，它们做一些运算并且因此会涉及到更多视图的使用。 这两个视图之一稍后将利用对 `INSERT`，`UPDATE`，`DELETE` 操作附加规则的方法自定义， 这样做最终的结果就是这个视图表现得像一个具有一些特殊功能的真正的表。 这个例子不适合于开始的简单易懂的例子，从这个例子开始讲可能会让讲解变得有些难以理解。 但是用一个覆盖所有关键点的例子来一步一步讨论要比举很多例子搞乱思维好。

比如，需要一个小巧的 `min` 函数用于返回两个整数值中较小的那个。用下面方法创建它：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

头两个规则系统要用到的表如下：

```
CREATE TABLE shoe_data (
    shoename   text,          -- 主键
    sh_avail   integer,       -- (鞋的)可用对数
    slcolor    text,          -- 首选的鞋带颜色
    slminlen   real,          -- 鞋带最短长度
    slmaxlen   real,          -- 鞋带最长长度
    slunit     text           -- 长度单位
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- 主键
    sl_avail   integer,       -- (鞋的)可用对数
    sl_color   text,          -- 鞋带颜色
    sl_len     real,          -- 鞋带长度
    sl_unit    text           -- 长度单位
);

CREATE TABLE unit (
    un_name    text,          -- 主键
    un_fact    real           -- 转换成厘米的系数
);
```

你可以看到，这些表代表鞋店的数据。

视图创建为：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

创建 `shoelace` 视图的 `CREATE VIEW` 命令(也是用到的最简单的一个) 将创建一个 `shoelace` 关系并且在 `pg_rewrite` 表里增加一个记录，告诉系统有一个重写规则应用于所有范围表里引用了 `shoelace` 关系的查询。该规则没有规则条件(将在非 `SELECT` 规则讨论，因为目前

的 `SELECT` 规则不可能有这些东西)并且它是 `INSTEAD` (取代)型的。要注意规则条件与查询条件不一样。 规则动作有一个查询条件。规则的动作是一个查询树，这个查询是树视图创建命令中的 `SELECT` 语句的一个拷贝。

> **Note:** 你在表 `pg_rewrite` 里看到的两个额外的用于 `NEW` 和 `OLD` 的范围表记录对 `SELECT` 规则不感兴趣。

现在填充 `unit` , `shoe_data` , `shoelace_data` ， 并且在视图上运行一个简单的查询：

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

这是可以在视图上做的最简单的 `SELECT` ，所以把它作为解释视图规则的基本命令。 `SELECT * FROM shoelace` 被分析器解释成下面的查询树:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

然后把这些交给规则系统。规则系统把范围表(range table)过滤一遍，检查一下有没有适用任何关系的规则。 当为 `shoelace` 记录处理范围表时(到目前为止唯一的一个)， 它会发现查询树里有 `_RETURN` 规则，查询树类似下面这样:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

为扩展该视图，重写器简单地创建一个子查询范围表记录，它包含规则动作的查询树， 然后用这个范围表记录取代原先引用视图的那个。生成的重写查询树几乎与你键入的那个一样：

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

不过还是有一个区别：子查询范围表有两个额外的记录 `shoelace old` 和 `shoelace new` 。这些记录并不直接参与查询， 因为它们没有被子查询的连接树或者目标列表引用。 重写器用它们存储最初出现在引用视图的范围表里面的访问权限检查。这样， 执行器仍然会检查该用户是否有访问视图的合适权限，即使在重写查询里面没有对视图的直接使用也如此。

这是应用的第一个规则。规则系统继续检查顶层查询里剩下的范围表记录(本例中没有了)，并且它在加进来的子查询中递归地检查范围表记录，看看其中有没有引用视图的 (不过这样不会扩展 `old` 或 `new` ，否则会无穷递归下去!)。 在这个例子中，没有用于 `shoelace_data` 或 `unit` 的重写规则， 所以重写结束并且上面的就是给规划器的最终结果。

现在想写这么一个查询：这个查询找出目前在店里有配对鞋带的鞋子（颜色和长度）， 并且配对的鞋带数大于或等于二。

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

这回分析器的输出是查询树:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

应用的第一个规则将是用于 `shoe_ready` 视图的，结果是生成查询树:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

与上面类似，用于 `shoe` 和 `shoelace` 的规则替换到子查询范围表里，生成一个最终的三层查询树：

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               min(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

最后规划器会把这个树压缩成一个两层查询树：最下层的 `SELECT` 将"拖到"中间的 `SELECT` 中，因为没有必要分别处理它们。 但是中间的 `SELECT` 仍然和顶层的分开，因为它包含聚集函数。 如果把它们也拉进来，那它就会修改最顶层 `SELECT` 的行为， 那可不是想要的。不过，压缩查询树是重写系统自己不需要关心的优化操作。

38.2.2. 非 `SELECT` 语句的视图规则

有两个查询树的细节在上面的视图规则中没有涉及到。就是命令类型和结果关系。实际上，视图规则不需要命令类型，但是结果关系可能会影响查询重写的工作方式， 因为如果结果关系是一个视图则需要特别的注意。

一个 `SELECT` 的查询树和用于其它命令的查询树只有少数几个区别。显然， 它们的命令类型不同并且对于 `SELECT` 之外的命令， 结果关系指向结果将前往的范围表入口。任何其它东西都完全是一样的。 所以如果有两个表 `t1` 和 `t2` 分别有字段 `a` 和 `b` ， 下面两个语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表包含表 `t1` 和 `t2` 的记录。
- 目标列表包含一个变量，该变量指向表 `t2` 的范围表入口的 `b` 字段。
- 条件表达式比较两个范围的字段 `a` 以寻找相等行。
- 连接树显示 `t1` 和 `t2` 之间的简单连接。

结果是，两个查询树生成相似的执行规划：它们都是两个表的连接。对于 `UPDATE` 语句来说，规划器把 `t1` 缺失的字段追加到目标列因而最终查询树看起来像：

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因此执行器在连接上运行的结果和下面语句是完全一样的：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是在 `UPDATE` 里有点问题：做链接的执行器计划部分不关心连接结果的含义是什么。 它只是产生一个行的结果集。一个是 `SELECT` 命令而另一个是 `UPDATE` 命令实际是在执行器的更高级处理的， 这里知道这是一个 `UPDATE` ，而且它还知道结果要记录到表 `t1` 里去。 但是现有的记录中的哪一行要被新行取代呢?

要解决这个问题，在 `UPDATE` 和 `DELETE` 语句的目标列表里面增加了另外一个入口：当前的行ID (CTID)。 这是一个有着特殊特性的系统字段。它包含行在文件块中的块编号和位置信息。 在已知表的情况下， 可以通过CTID检索最初的需要更新的 `t1` 行。 在把CTID加到目标列表中去以后，查询看上去实际上像这样：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在，另一个PostgreSQL的细节进入到这个阶段里了。这时， 表中的旧行还没有被覆盖，这就是为什么 `ROLLBACK` 飞快的原因。 在一个 `UPDATE` 里，新的结果行插入到表里(在剥除CTID之后) 并且把CTID指向的旧数据行的行头里面的 `cmax` 和 `xmax` 设置为当前命令计数器和当前事务 ID 。这样旧的行就被隐藏起来并且在事务提交之后， vacuum 清理器就可以真正把它们删除掉。

知道了这些，就可以简单的把视图的规则应用到任意命令中。规则和命令没有区别。

38.2.3. PostgreSQL里视图的强大能力

上面演示了规则系统如何融合视图定义到初始查询树中去。在第二个例子里， 一个简单的对视图的 `SELECT` 创建了一个四表联合的查询树 (`unit` 以不同的名称用了两次)。

在规则系统里实现视图的好处是，规划器在一个查询树里拥有所有信息： 应该扫描哪个表+表之间的关系+视图的资格限制+初始查询的资格(条件)。 并且仍然是在最初的查询已经是一个视图的联合的情况下。现在规划器必须决定执行查询的最优路径。 规划器拥有越多信息，它的决策就越好。并且PostgreSQL 里的规则系统的实现保证这些信息是此时能获得的有关该查询的所有信息。

38.2.4. 更新一个视图

如果视图命名为 `INSERT` , `UPDATE` , `DELETE` 的目标关系会怎样?在完成上面描述的替换之后，就有一个这样的查询树： 结果关系指向一个是子查询的范围表记录，这样可不能运行。 在PostgreSQL中有几种方式支持更新一个视图的外观。

如果子查询从一个单一的基本关系选择或者足够简单，重写可以自动的用底层的基本关系替代子查询， 所以 `INSERT` ， `UPDATE` 或 `DELETE` 以适当的方式应用于基本关系。视图因为"足够简单"而被称为可自动更新的。 有关这种可以自动更新的视图的更详细的信息请参阅CREATE VIEW。

或者，操作可能通过一个用户提供的在视图上的 `INSTEAD OF` 触发器处理。 重写工作在这种情况下略有不同。对于 `INSERT` ， 重写并不对视图做什么，让它作为查询的结果关系。对于 `UPDATE` 和 `DELETE` ， 重写仍然需要扩展视图查询，产生命令将要更新或删除的"old" 行。所以，视图正常扩展，但是另外一个不扩展的范围表条目添加到查询中， 代表视图作为结果关系。

现在出现的问题是如何识别出视图中要更新的行。回想当结果关系是一个表时， 一个特殊的CTID条目被添加到目标列表，以识别出要被更新的行的物理位置。 如果结果关系是一个视图这将不会工作，因为视图没有任何CTID， 因为它的行没有真实的物理位置。相反，对于 `UPDATE` 或 `DELETE` 操作，一个特殊的 `wholerow` 条目被添加到目标列表，它扩大到包含视图的所有列。执行器使用这个值提供"old" 行到 `INSTEAD OF` 触发器。由触发器基于旧行和新行值找出来的需要更新什么。

另外一个可能是用户定义`INSTEAD`规则，为视图上的`INSERT`, `UPDATE`, 和 `DELETE`命令指定替代动作。 这些规则将重写命令，通常进入一个命令更新一个或更多的表，而不是视图。 这是下一节的主题。

请注意，首先评估规则，在规划和执行之前重写原先的查询。因此， 如果一个视图有`INSTEAD OF`触发器，也有在`INSERT`, `UPDATE`, 或 `DELETE`上的规则，那么规则将被首先评估， 根据评估结果，触发器可能不会使用。

在一个简单视图上的`INSERT`, `UPDATE`, 或 `DELETE`查询的自动重写总是最后尝试。因此， 如果一个视图有规则或触发器，他们将覆盖自动更新视图的缺省行为。

如果视图上没有`INSTEAD`规则或`INSTEAD OF`触发器， 并且重写不能作为一个在底层基本关系上的更新自动重写查询， 那么将抛出一个错误，因为执行器不能像这样的更新视图。

38.3. 物化视图

PostgreSQL里的物化视图像视图那样使用规则系统， 但是用类表的形式保存结果。

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

和：

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

之间最主要的区别是物化视图不能随后直接被更新，并且创建物化视图的查询就像视图的查询存储那样存储， 所以新数据可以用下面命令产生：

```
REFRESH MATERIALIZED VIEW mymatview;
```

PostgreSQL系统目录中有关物化视图的信息和表或视图的信息一样。 所以对于解析器，物化视图是一个关系，就像一个表或一个视图。当在查询中引用一个物化视图时， 数据直接从物化视图返回，就像从一个表返回；规则只是用来填充物化视图。

当访问存储在物化视图中的数据时，通常比直接访问底层表或通过一个视图更快， 数据并不总是当前的；然而有时不需要当前数据。考虑一个记录销售的表：

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- 销售人员的ID
    invoice_date  date,          -- 销售日期
    invoice_amt   numeric(13,2)  -- 销售数量
);
```

如果人们希望能够快速的图形化历史销售数据，他们可能想要汇总， 可能不关心当前未完成的数据：

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date
    ORDER BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

物化视图可以用来在为销售人员创建的控制面板上显示图形。 可以使用下面的SQL语句在每天晚上更新统计数据：

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一个用处是允许对远程系统中的数据快速访问，通过一个外部数据封装器。 下面是一个简单的使用 `file_fdw` 的例子，有计时， 但是因为这是使用的在本地系统的缓存，外部数据封装器到远程系统的性能可能更大。

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/etc/dictionaries-common/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

现在让我们拼写检查一个单词。直接使用 `file_fdw` ：

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

计划是：

```
 Aggregate  (cost=4125.19..4125.20 rows=1 width=0) (actual time=26.013..26.014 rows=1 loo
   ->  Foreign Scan on words  (cost=0.00..4124.70 rows=196 width=0) (actual time=26.011..
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 99171
         Foreign File: /etc/dictionaries-common/words
         Foreign File Size: 938848
 Total runtime: 26.081 ms
```

如果使用物化视图，查询更快速：

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.074..0.074 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual ti
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Total runtime: 0.119 ms
```

无论哪种方式，这个词的拼写是错误的，所以我们看看我们想要的。还是使用 `file_fdw` ：

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=2195.70..2195.72 rows=10 width=32) (actual time=218.904..218.906 rows=10 lo
   ->  Sort  (cost=2195.70..2237.61 rows=16765 width=32) (actual time=218.902..218.904 ro
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..1833.41 rows=16765 width=32) (actual time
               Foreign File: /etc/dictionaries-common/words
               Foreign File Size: 938848
 Total runtime: 218.966 ms
```

使用物化视图：

```
 Limit  (cost=0.28..1.02 rows=10 width=9) (actual time=24.916..25.079 rows=10 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.28..7383.70 rows=99171 width=9) (actual
         Order By: (word <-> 'caterpiler'::text)
 Total runtime: 25.884 ms
```

如果你能允许定期更新远程数据到本地数据库，会带来可观的性能优势。

38.4. 在 `INSERT`, `UPDATE`, 和 `DELETE` 上的规则

定义在 `INSERT`, `UPDATE`, `DELETE` 上的规则与前一章描述的视图规则完全不同。首先，他们的 `CREATE RULE` 命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 他们可以是 `INSTEAD` 或 `ALSO` (缺省)。
- 伪关系 `NEW` 和 `OLD` 变得有用了。
- 它们可以有规则资格条件。

第二，它们不是就地修改查询树，而是创建零个或多个新查询树并且可能把原始的那个扔掉。

38.4.1. 更新规则是如何运转的

把下面语法：

```
CREATE [ OR REPLACE ] RULE _name_ AS ON _event_
    TO _table_ [ WHERE _condition_ ]
    DO [ ALSO | INSTEAD ] { NOTHING | _command_ | ( _command_ ; _command_ ... ) }
```

牢牢记住。在随后的内容里，*update rules*(更新规则)意思是定义在 `INSERT`, `UPDATE`, 或 `DELETE` 上的规则。

如果查询树的结果关系和命令类型与 `CREATE RULE` 命令里给出的对象和事件一样的话， 规则系统就把更新规则应用上去。对于更新规则，规则系统创建一个查询树列表。 一开始查询树是空的，这里可以有零个(`NOTHING` 关键字)、一个、或多个动作。 为简单起见，先看一个只有一个动作的规则。这个规则可以有零个或一个条件并且它可以是 `INSTEAD` 或 `ALSO` (缺省)。

何为规则条件?它是一个限制条件，告诉规则动作什么时候要做，什么时候不要做。 这个条件可以只引用 `NEW` 和/或 `OLD` 伪关系， 它们基本上是代表以对象形式给出的基本关系(但是有着特殊含义)。

所以，对这个单动作的规则生成查询树，有下面三种情况。

没有条件，也没有 `ALSO` 或 `INSTEAD`

来自规则动作的查询树，附加了原始查询树的条件。

给出了条件，有 `ALSO`

来自规则动作的带有规则条件的查询树并且附加了原始查询树的条件。

给出了条件，有 `INSTEAD`

来自规则动作带有规则条件的查询树以及原始查询树的条件；以及附加了相反规则条件的原始查询树。

最后，如果规则是 `ALSO`，那么最初未修改的查询树被加入到列表。 因为只有合格的 `INSTEAD` 规则已经在初始的查询树里面， 所以对于单动作规则最终得到一个或者两个查询树。

对于 `ON INSERT` 规则，原来的查询(如果没有被 `INSTEAD` 取代) 是在任何规则增加的动作之前完成的。这样就允许动作看到插入的行。但是对 `ON UPDATE` 和 `ON DELETE` 规则，原来的查询是在规则增加的动作之后完成的。 这样就确保动作可以看到将要更新或者将要删除的行；否则， 动作可能什么也不做， 因为它们发现没有符合它们要求的行。

从规则动作生成的查询树被再次送到重写系统，并且可能附加更多的规则，结果是更多的或更少的查询树。 所以规则动作必须是另一个命令类型或者和规则所在的关系不同的另一个结果关系。 否则这样的递归过程就会没完没了(规则的递规展开会被检测到，并当作一个错误报告)。

在 `pg_rewrite` 系统表中 action 里的查询树只是模板。 因为他们可以引用范围表的 `NEW` 和 `OLD`，在使用它们之前必须做一些调整。 对于任何对 `NEW` 的引用，都要先在初始查询的目标列中搜索对应的条目。如果找到， 把该条目表达式放到引用里。否则 `NEW` 和 `OLD` 的含义一样(对于 `UPDATE`) 或者被 NULL 替代(对于 `INSERT`)。任何对 `OLD` 的引用都用结果关系的范围表的引用替换。

在系统完成更新规则的附加之后，它再附加视图规则到生成的查询树上。视图无法插入新的更新动作， 所以没有必要向视图重写的输出附加更新规则。

38.4.1.1. 循序渐进的第一个规则

假设希望跟踪 `shoelace_data` 关系中的 `sl_avail` 字段。 所以设置一个日志表和一条规则，这条规则每次在用 `UPDATE` 更新 `shoelace_data` 表时都要往数据库里写一条记录。

```
 CREATE TABLE shoelace_log (
    sl_name    text,          -- 鞋带变化了
    sl_avail   integer,       -- 新的可用数值
    log_who    text,          -- 谁干的
    log_when   timestamp      -- 什么时候
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

现在有人键入：

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

然后看看日志表：

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

这是想要的。后端发生的事情如下。分析器创建查询树：

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

这里是一个带有条件表达式的 `ON UPDATE` 规则 `log_shoelace` ：

```
NEW.sl_avail <> OLD.sl_avail
```

和动作：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

这个输出看起来有点奇怪，因为你不能写 `INSERT ... VALUES ... FROM` 。 这里的 `FROM` 子句只是表示查询树里有用于 `new` 和 `old` 的范围表记录。这些东西的存在是因为这样一来它们就可以被 `INSERT` 命令的查询树里的变量引用。

该规则是一个有条件的 `ALSO` 规则，所以规则系统必须返回两个查询树：更改过的规则动作和原始查询树。在第一步里，原始查询的范围表集成到规则动作查询树里。生成：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       **shoelace_data shoelace_data**;
```

第二步把规则条件增加进去，所以结果集限制为 `sl_avail` 改变了的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 **WHERE new.sl_avail <> old.sl_avail**;
```

这个东西看起来更奇怪，因为 `INSERT ... VALUES` 也没有 `WHERE` 子句， 不过规划器和执行器对此并不在意。它们毕竟还要为 `INSERT ... SELECT` 支持这种功能。

第三步把原始查询树的条件加进去，把结果集进一步限制成只有被初始查询树改变的行：

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   **AND shoelace_data.sl_name = 'sl7'**;
```

第四步把 `NEW` 引用替换为从原始查询树来的目标列或从结果关系来的相匹配的变量引用：

```
INSERT INTO shoelace_log VALUES (
       **shoelace_data.sl_name**, **6**,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE **6** <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

第五步用结果关系引用把 `OLD` 引用替换掉：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> **shoelace_data.sl_avail**
   AND shoelace_data.sl_name = 'sl7';
```

这就成了。因为规则 `ALSO` 还输出原始查询树。简而言之， 从规则系统输出的是一个两个查询树的列表，与下面语句相同：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

这就是执行的顺序以及规则要做的事情。

做的替换和追加的条件用于确保如果原始的查询是下面这样：

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

就不会有日期记录写到表里。因为这回原始查询树不包含有关 `sl_avail` 的目标列表， `NEW.sl_avail` 将被 `shoelace_data.sl_avail` 代替， 所以，规则生成的额外命令是：

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, **shoelace_data.sl_avail**,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE **shoelace_data.sl_avail** <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

并且条件将永远不可能是真值。

如果最初的查询修改多个行，它也能运行。所以如果写出下面命令：

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

实际上有四行被更新(`sl1` , `sl2` , `sl3` , 和 `sl4`)。 但 `sl3` 已经是 `sl_avail = 0` 。这回，原始的查询树条件已经不一样了， 结果是规则生成下面的额外查询树：

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND **shoelace_data.sl_color = 'black'**;
```

这个查询树将肯定插入三个新的日志记录。这也是完全正确的。

到这里就明白为什么原始查询树最后执行非常重要。如果 `UPDATE` 将先被执行， 所有的行都已经设为零，所以记日志的 `INSERT` 将不能找到任何符合 `0 <> shoelace_data.sl_avail` 条件的行。

38.4.2. 与视图合作

一个保护视图关系，使其避免有人可以在其中 `INSERT`，`UPDATE`，`DELETE` 的简单方法是让那些查询树被丢弃。创建下面规则：

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

如果现在任何人试图对视图关系 `shoe` 做上面的任何操作， 规则系统将应用这些规则。因为这些规则没有动作而且是 `INSTEAD`， 结果是生成的查询树将是空的并且整个查询将变得空空如也， 因为经过规则系统处理后没有什么东西剩下来用于优化或执行了。

一个更复杂的使用规则系统的方法是用规则系统创建一个重写查询树的规则， 使查询树对真实的表进行正确的操作。要在视图 `shoelace` 上做这个工作，创建下面规则：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

如果你打算在视图上支持 `RETURNING` 查询，就要让规则包含 `RETURNING` 计算视图行数的子句。这对于基于单个表的视图来说通常非常琐碎，但是连接诸如 `shoelace` 之类的视图很单调乏味。一个插入情况的例子如下：

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

注意，这个规则同时支持该视图上的 `INSERT` 和 `INSERT RETURNING` 查询， `INSERT` 将简单的忽略 `RETURNING` 子句。

假设现在有一包鞋带到达商店，还有一个大的部件列表。但是不想每次都手工更新 `shoelace` 视图。取而代之的是创建了两个小表： 一个是可以从到货清单中插入东西， 另一个是一个特殊的技巧。创建这些的命令如下：

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

现在你可以用来自部件列表的数据填充表 `shoelace_arrive` 了：

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

让我们迅速地看一眼当前的数据，

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

把到货鞋带移到(shoelace_ok)中：

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

然后检查结果：

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从 `INSERT ... SELECT` 语句到这个结果经过了长长的一段过程。 而且对查询树转化的描述将是本文档的最后。首先是生成分析器输出：

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则 `shoelace_ok_ins` 把它转换成：

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

并且把原始的对 `shoelace_ok` 的 `INSERT` 丢弃掉。 这样重写后的查询再次传入规则系统并且第二次应用了规则 `shoelace_upd` 生成：

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

同样这是一个 `INSTEAD` 规则并且前一个查询树被丢弃掉。 注意这个查询仍然是使用视图 `shoelace` ，但是规则系统还没有完成这一步， 所以它继续在这上面应用规则 `_RETURN` ，然后得到：

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

最后，应用规则 `log_shoelace` ，生成额外的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

这样，在规则系统用完所有的规则后返回生成的查询树。

所以最终得到两个等效于下面SQL语句的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

结果是从一个关系来的数据插入到另一个中，到了第三个中变成更新， 在到第四个中变成更新加上记日志，最后在第五个规则中缩减为两个查询。

有一个小细节有点让人难受。看看生成的两个查询，会发现 `shoelace_data` 关系在范围表中出现了两次而实际上绝对可以缩为一次。因为规划器不处理这些， 所以对规则系统输出的 `INSERT` 的执行规划会是

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

在省略多余的范围表后的结果将是

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

这也会在日志关系中生成完全一样的记录。因此，规则系统导致对表 `shoelace_data` 的一次多余的扫描，而且同样多余的扫描会在 `UPDATE` 里也一样多做一次。 不过要想把这些不足去掉是一样太困难的活了。

最后对PostgreSQL规则系统及其功能做一个演示。 假设你向你的数据库中添加一些比较罕见的鞋带：

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

建立一个视图检查哪种 `shoelace` 记录在颜色上和任何鞋子都不相配。 用于这个的视图是：

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

现在想这样设置：没有库存的不匹配的鞋带都从数据库中删除。为了让这事对 PostgreSQL有点难度，不直接删除它们。 取而代之的是再创建一个视图:

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后用下面方法做：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

所以à:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

对一个视图的 `DELETE` ，这个视图带有一个总共使用了四个嵌套/连接的视图的子查询条件，这四个视图之一本身有一个拥有对一个视图的子查询条件，该条件计算使用的视图的列；最后重写成了一个查询树，该查询树从一个真正的表里面把需要删除的数据删除。

我想在现实世界里只有很少的机会需要上面的这样的构造。但这些东西能运转肯定让你舒服。

38.5. 规则和权限

由于PostgreSQL规则系统对查询的重写， 非初始查询指定的其它表/视图被访问。使用更新规则的时候，这可能包括对表的写权限。

重写规则并不拥有一个独立的所有者。关系(表或视图)的所有者自动成为重写规则的缺省所有者。 PostgreSQL规则系统改变缺省的访问控制系统的特性。 因规则而使用的关系要对定义规则所有者进行权限检查，而不是激活规则的用户， 这意味着一个用户只需要对他的查询里明确指定的表/视图拥有所需的权限就可进行操作。

例如：某用户有一个电话号码列表，其中一些是私人的，另外的一些是办公室秘书需要的。他可以用下面方法构建查询：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

除了他以外(还有数据库超级用户)没有人可以访问 `phone_data` 表。 但因为 `GRANT` 的原因，秘书可以从 `phone_number` 视图上运行 `SELECT` 。规则系统将把从 `phone_number` 里的 `SELECT` 重写为从 `phone_data` 里的 `SELECT` 。因为用户是 `phone_number` 的所有者， 因此也是规则的所有者，所以现在要检查他对 `phone_data` 的读访问的权限， 而这个查询是被允许的。同时也要检查访问 `phone_number` 的权限， 但这是对一个被撤消权限的用户进行检查的，所以除了用户自己和秘书外没有人可以使用它。

权限检查是按规则逐条进行的。所以此时的秘书是唯一的一个可以看到公共电话号码的人。但秘书可以设立另一个视图并且赋予该视图公共权限。这样， 任何人都可以通过秘书的视图看到 `phone_number` 数据。 秘书不能做的事情是创建一个直接访问 `phone_data` 的视图(实际上他是可以的， 但没有任何作用，因为每个访问都会因通不过权限检查而被踢出事务)。而且用户很快会认识到， 秘书开放了他的 `phone_number` 视图后，他还可以撤消他的访问权限。这样， 所有对秘书视图的访问马上就失效了。

有些人会认为这种逐条规则的检查是一个安全漏洞，但事实上不是。如果这样做不能奏效，秘书将必须建立一个与 `phone_number` 有相同字段的表并且每天拷贝一次数据进去。 那么这是他自己的数据因而可以赋予其它人访问的权力。一个 `GRANT` 意味着"我信任你"。如果某个你信任的人做了上面的事情， 那你就该想想是否该 `REVOKE` 了。

请注意，当视图可以用来隐藏使用了上面的技术的特定的内容时， 它们不能用来可靠地隐藏在不可视的行的数据，除非已经设置了 `security_barrier` 标志。例如，下面的视图是不安全的：

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图可能看起来是安全的，因为规则系统将重写 `phone_number` 里面的任意 `SELECT` 到 `phone_data` 里面的 `SELECT` ， 并且添加只有不是以412开头的 `phone` 才是所需要的条目的条件。 但是如果用户可以创建他或她自己的函数，那么使规划器在 `NOT LIKE` 表达式之前执行用户定义的函数也不困难。例如：

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

`phone_data` 表里的每个人和电话号码将作为一个 `NOTICE` 打印， 因为规划器将在更多的昂贵的 `NOT LIKE` 之前执行 廉价的 `tricky` 函数。即使阻止了用户定义新的函数， 内建的函数可以用在类似的安全系统里。（例如， 大多数转换函数在他们产生的错误信息中包含他们的输入值。）

类似的注意事项应用于更新规则。在上一章的例子里，例子数据库里的表的所有者可以把 `shoelace` 视图的 `SELECT` ，`INSERT` ，`UPDATE` ，`DELETE` 权限赋予其他人。但对 `shoelace_log` 只有 `SELECT` 权限。 写日志记录的规则动作仍然可以成功的执行。并且其它用户可以看到日志记录。 但他不能创建伪记录，而且他也不能对现有记录进行修改或删除。 在这种情况下，不能通过使规划器改变操作的顺序来推翻规则， 因为唯一引用 `shoelace_log` 的规则是一个未修饰的 `INSERT` 。 在更复杂的情形中这可能不是真的。

当必须为视图提供行级别的安全性时， `security_barrier` 属性应该应用于视图。这样就阻止了在行上调用恶意的选择函数和操作，直到视图完成了它的工作。 例如，如果上面显示的视图像这样创建，它将是安全的：

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

带有 `security_barrier` 创建视图可能执行起来更糟糕。 通常，这是没有办法避免的：如果可能危及安全，那么最快可能的计划必须拒绝。 因为这个原因，缺省不启用这个选项。

当处理的函数没有副作用时，查询计划器有更大的灵活性。这样的函数被称为 `LEAKPROOF` ，包括许多简单、常用的操作符，例如许多相等操作符。查询计划器可以安全的允许这样的函数在查询执行过程的任何时候评估。 因为在行上不可见的调用它们不会给用户泄露任何关于不可见行的信息。相反的， 一个函数可能会抛出一个错误，取决于作为参数（例如在溢出或除以零的情况下抛出一个错误） 接收到的值是非防止泄露的，并且如果在安全视图的行过滤器之前应用可能提供重要的关于不可见行的信息。

理解这个是重要的，即使视图创建时带有了 `security_barrier` 选项， 狭义来讲，为了安全，只有不可见元组的内容不会被传送到可能不安全的函数。 用户很可能对于推断不可见的数据有其他的意思；例如，他们使用 `EXPLAIN` 看到查询计划，或测量查询在视图上的运行时间。恶意攻击者也许能够推断不可见数据的数量， 或者甚至获得一些关于数据分布或常见值的信息（因为这些事情可能影响计划的运行时间； 甚至，因为他们也反映在优化器的统计数据中，计划的选择）。如果关心了这些"covert channel" 类型的攻击，那么给这些数据授予任何权限可能是不明智的。

38.6. 规则和命令状态

PostgreSQL服务器为它收到的每个命令返回一个命令状态字符串， 比如 `INSERT 149592 1` 。如果没有涉及规则，那么这些就很简单， 但是如果查询是被规则重写的又会怎样呢?

规则对命令状态的影响如下：

- 如果查询不存在无条件的 `INSTEAD` 规则，那么最初给出的查询将会被执行， 并且它的命令状态将像平常一样返回。但是请注意如果存在任何条件 `INSTEAD` 规则， 那么他们的条件的反条件将会已经加到最初的查询里了。这样可能会减少它处理的行数， 如果这样的话，报告状态将受影响。

- 如果查询有任何无条件的 `INSTEAD` 规则，那么最初的查询将完全不会被执行。 在这种情况下，服务器将返回由 `INSTEAD` 规则(条件的或非条件的) 插入的最后一条和源查询同命令类型(`INSERT` , `UPDATE` , 或 `DELETE`)查询的命令状态。如果规则添加的查询都不符合这些要求， 那么返回的命令状态显示源查询类型而行计数和 OID 字段为零。

（这个系统是在PostgreSQL 7.3建立的。在之前的版本里， 命令状态可能在规则退出时显示不同的结果。）

程序员可以用下面的方法确保任何需要的 `INSTEAD` 规则都是在上面第二种情况里设置命令状态的规则： 给这个规则命名为字母顺序最后一个活动的规则，这样它就最后附加。

38.7. 规则与触发器的比较

许多用触发器可以干的事情同样也可以用PostgreSQL规则系统来实现。 目前不能用规则来实现的东西之一是某些约束，特别是外键。 可能在某字段的值没有在另一个表里出现的情况下用一条有条件的规则把查询重写为 `NOTHING` 。 不过这样做数据就会被不声不响的扔掉，因而这也不是一个好主意。如果需要检查有效的值， 而且如果是无效值出现时要生成一个错误信息，这种情况下要用触发器来做。

在本节中，我们专注于使用规则更新视图。本节中的所有更新规则的示例也可以使用在视图上的 `INSTEAD OF` 触发器实现。编写这样的触发器通常比编写规则要容易， 尤其是需要复杂的逻辑执行更新时。

对于两者都可用的情况，哪个更好取决于对数据库的使用。触发器为每个涉及到的行执行一次。 规则修改查询或生成额外的查询。所以如果在一个语句中涉及到多行， 一个生成额外查询的规则通常可能会比一个对每一行都分别执行一次(而且要重新决定做什么很多次) 的触发器快一些。不过，触发器的方法从概念上要远比规则的方法简单，并且很容易让新手可以做正确事情。

下面展示一个在同一个情况下选择规则与触发器的对比例子。例如这里有两个表：

```
CREATE TABLE computer (
hostname        text,    -- 已索引
    manufacturer    text     -- 已索引
);

CREATE TABLE software (

software        text,    -- 已索引
    hostname        text     -- 已索引
);
```

两个表都有好几千行，并且 `hostname` 上的索引是唯一的。规则/触发器应该实现这样一个约束， 这个约束从 `software` 表中删除引用已删除计算机的行。触发器可以用下面这条命令：

```
DELETE FROM software WHERE hostname = $1;
```

因为触发器是为从 `computer` 里面删除的每一个独立的行调用一次， 那么它可以准备并且保存这个命令的规划，把 `hostname` 作为参数传递。 规则应该这样写：

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在看看这两种不同的删除。在下面情况：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

对表 `computer` 使用索引(快速)进行扫描并且由触发器声明的查询也用索引进行扫描(同样快速)。 规则里多出来的查询是一个：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

因为已经建立了合适的索引，规划器将创建一个下面的规划

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

所以在规则和触发器的实现之间没有太多的速度差别。

下面的删除希望删掉所有 2000 个 `hostname` 以 `old` 开头的计算机(记录)。 有两个可能的用于这个用途的查询。一个是：

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

规则增加的命令是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

查询的规划将会是

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

另一个可能的查询是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它由规则增加执行规划是：

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

这表明，规划器不能认识到表 `computer` 里的 `hostname` 的条件在多个条件表达式以 `AND` 的方式组合在一起时同样可以用于 `software` 上的索引扫描， 就像在用正则表达式的查询里一样。触发器将在任何 2000 个要被删除的旧计算机里被调用一次， 结果是对 `computer` 的一次索引扫描和对 `software` 的 2000 次索引扫描。 规则的实现将会使用两个使用索引的命令来完成。 所以 `software` 表的实际大小决定了规则进行顺序扫描后是否仍然更快。即使所有要使用的索引块都很快在缓冲里出现， 执行 2000 个在 SPI 管理器上的查询仍然要花不少时间。

我们要看的最后一个查询是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致从 `computer` 表里删除多行。 所以触发器同样会向执行器提交很多查询。规则生成的命令将会是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

但规则规划又将是对两个索引扫描的嵌套循环。不过使用了 `computer` 的另外一个索引：

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

在任何一种情况下，从规则系统出来的额外查询都或多或少与查询中涉及到的行数量相对独立。

概括来说，规则只是在它们的动作生成了又大又烂的条件连接时才比触发器有较大速度差异， 这时规划器将失效。

Chapter 39. 过程语言

PostgreSQL允许用户定义的函数使用SQL和C之外的语言编写。 通常这些额外的语言叫过程语言(PLs)。 如果用一种过程语言书写了一个函数， 那么数据库服务器没有任何内建的办法解析该函数的源文本。 实际上这些任务都传递给一个知道如何处理这些细节的处理器处理。 这个处理器既可以自己做所有的分析,语法分析， 执行等工作， 也可以充当PostgreSQL和一种现有的编程语言实现之间的"胶水"。 处理器本身是一个 C 语言函数，它被编译成共享对象并且在需要的时候加载， 就像其它 C 函数一样。

目前在标准的PostgreSQL发布里有四种过程语言可用： PL/pgSQL (Chapter 40), PL/Tcl (Chapter 41), PL/Perl (Chapter 42)和 PL/Python (Chapter 43)。 还有几种额外的过程语言没有包含在核心发布里。Appendix H里面有如何找到它们的信息。 用户可以定义其它语言。开发一种新的过程语言的基本信息在Chapter 51里介绍。

39.1. 安装过程语言

如果你要使用某种过程语言，那么你必须把它"安装"到要使用它的数据库里。 不过那些安装到数据库 `template1` 里的过程语言会自动在随后创建的数据库中安装。 因为 `template1` 中的目录通过 `CREATE DATABASE` 进行拷贝， 因此数据库管理员可以决定哪个数据库可以使用哪种语言， 以及决定缺省时可以使用哪些语言。

对于那些随着标准版本发布的语言， 只需要使用 `CREATE EXTENSION` `_language_name_` 把语言安装到当前数据库中即可。 另外，也可以使用createlang程序在shell命令行上安装语言。 比如，要将PL/Perl安装到 `template1` 数据库中，可以使用：

```
createlang plperl template1
```

推荐下面描述的手工程序来安装没有作为扩展包装的定制语言。

手工安装过程语言

一个过程语言是按五个步骤安装到数据库里面去的， 这些任务必须由数据库超级用户执行。 在大多数情况下所需的SQL命令应该被打包成"扩展"的安装脚本， 因此 `CREATE EXTENSION` 可以用于执行它们。

1. 必须编译和安装该语言处理器的共享对象并安装到一个合适的库目录中。 方法和安装用户定义的 C 函数的方法(Section 35.9.6)一样。通常， 语言处理器需要外部库提供实际的引擎；如果是这样， 那么这些库也必须安装。

2. 处理器必须使用下面的命令声明

   ```
   CREATE FUNCTION _handler_function_name_()
       RETURNS language_handler
       AS '_path-to-shared-object_'
       LANGUAGE C;
   ```

   `language_handler` 的返回类型告诉数据库系统该函数并不返回任何已定义的SQL数据类型， 并且不能在SQL语句中被直接使用。

3. 可选步骤，语言处理器可以提供一个"内置"函数， 这个函数执行使用这种语言编写的匿名代码块（DO命令）。 如果通过该语言提供一个内置处理函数，那么用下面的命令声明：

   ```
   CREATE FUNCTION _inline_function_name_(internal)
       RETURNS void
       AS '_path-to-shared-object_'
       LANGUAGE C;
   ```

4. 可选步骤，语言处理器可以提供一个"验证器"函数， 这个函数为没有实际执行它的正确性而检查函数定义。 如果存在的话，通过 `CREATE FUNCTION` 调用验证函数。 如果通过这种语言提供了验证函数，那么用下面的命令声明

```
CREATE FUNCTION _validator_function_name_(oid)
    RETURNS void
    AS '_path-to-shared-object_'
    LANGUAGE C STRICT;
```

5. 最后，PL必须通过下面的命令声明

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE _language-name_
    HANDLER _handler_function_name_
    [INLINE `_inline_function_name_`]
    [VALIDATOR `_validator_function_name_`] ;
```

   可选关键字 `TRUSTED` 指明该语言不授予访问用户不拥有的数据的权限， 为一个没有超级用户权限的普通数据库用户设计可信任语言， 并且允许他们安全地使用这种语言创建函数和触发器过程。 因为 PL 函数在数据库服务器中执行， 所以 `TRUSTED` 标志应该只是用于那些不允许访问数据库服务器内部或文件系统的语言。 PL/pgSQL, PL/Tcl和 PL/Perl语言都是可信的。PL/TclU, PL/PerlU和 PL/PythonU 都被设计成提供无限的功能，因此不应该标记为可信。

Example 39-1显示了如何手工安装PL/Perl语言。

Example 39-1. 手工安装PL/Perl

下面的命令告诉数据库服务器哪里才能找到用于PL/Perl语言的调用处理器函数的共享对象。

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl有一个内置处理函数和验证函数，因此我们也要声明它们:

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

命令：

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

声明了前面所定义的函数应该被那些调用语言属性是 `plperl` 的 函数或触发器过程使用。

在缺省的PostgreSQL安装里， PL/pgSQL语言的处理器是制作并安装到"library"目录中去的。进一步来说，PL/pgSQL语言自身安装到所有数据库中， 如果配置了Tcl支持，那么PL/Tcl和PL/TclU 的处理器也都被编译并安装到同一个库目录中。 但是语言自身缺省不安装到任何数据库中， 类似的， 如果配置了Perl支持，则PL/Perl和PL/PerlU的处理器也都编译并且安装。并且如果配置了Python支持，则安装PL/PythonU处理器。 但是这些语言缺省不安装。

Chapter 40. PL/pgSQL - SQL过程语言

Table of Contents

40.1. 概述

PL/pgSQL是PostgreSQL数据库系统的一个可加载的过程语言。 PL/pgSQL的设计目标是创建一种可加载的过程语言，可以

- 用于创建函数和触发器过程，
- 为SQL语言增加控制结构，
- 执行复杂的计算，
- 继承所有用户定义类型、函数、操作符，
- 定义为被服务器信任的语言，
- 容易使用。

PL/pgSQL创建的函数可以在那些使用内置函数一样的情形下使用。 比如，可以创建复杂的条件计算函数，并随后将之用于定义操作符或者用于函数索引中。

在PostgreSQL 9.0及其之后的版本中，PL/pgSQL是默认安装的。 当然，PL/pgSQL仍然是一个可加载的模块，因此，如果实际安全需要，管理员也可以选择将它卸载掉。

40.1.1. 使用PL/pgSQL的优点

SQL是PostgreSQL和大多数其它关系型数据库的命令语言。 它是可移植的，并且容易学习使用。但是所有SQL语句都必须由数据库服务器独立地执行。

这就意味着你的客户端应用必须把每条命令发送到数据库服务器， 等待它处理这个命令，接收结果，对结果进行一些处理， 然后再给服务器发送另外一条命令。 所有这些东西都会产生进程间通讯， 并且如果你的客户端在另外一台机器上甚至还会产生网络开销。

通过PL/pgSQL，可以把运算块和一系列命令在数据库服务器内部组成一个块， 这样就拥有了过程语言的能力并且简化 SQL 的使用， 因而节约了大量的时间，因为不需要进行客户端/服务器通讯。

- 忽略了客户端和服务器端之间的额外往返行程。
- 客户端不需要的中间结果无需在服务器端和客户端来回传递。
- 不需要多次语法分析步骤。

比起不使用存储函数来，这样做能够产生明显的性能提升。

同样，在PL/pgSQL里，仍然可以使用SQL的所有数据类型，操作符和函数。

40.1.2. 支持的参数和结果数据类型

使用PL/pgSQL所写的函数能够接受服务器支持的任何标量或数组数据类型作为参数， 并且同样能够返回这些类型的结果， 它们还可以接受或者返回任意用名字声明的复合类型(行类型)。 还可以将一个PL/pgSQL函数声明为一个返回 `record` 类型(行类型)， 表明该结果是一个行类型，这个行的字段是在调用它的查询中指定的， 就像在Section 7.2.1.4里讨论的那样。

与声明SQL函数一样，通过使用 `VARIADIC` 可以对PL/pgSQL 进行声明为能接受可变数目的参数。 正如在Section 35.4.5中讨论的那样。

PL/pgSQL函数还可以声明为接受并返回多态的 `anyelement` , `anyarray` , `anynonarray` , `anyenum` 和 `anyrange` 类型。 一个多态函数实际操作的数据类型可以在不同的调用环境中变化， 如在Section 35.2.5里讨论的那样。 Section 40.3.1是一个使用例子。

PL/pgSQL还可以声明为任何一个单个实例返回的数据类型"set"，或者表。 这样的函数通过为结果集每个需要返回的元素执行一个 `RETURN NEXT` 生成它的输出， 或者通过使用 `RETURN QUERY` 来输出评估查询的结果。

最后，如果返回的结果没有太大的价值，PL/pgSQL函数可以声明为返回 `void` 。

PL/pgSQL函数也可以声明为输出某种类型的参数， 来代替明确的返回类型声明。 这么做并未给该语言增加任何基础设施， 只是通常更方便些，特别是返回多行数值的时候。同时， 也可以用 `RETURNS TABLE` 来代替 `RETURNS SETOF` 。

具体的例子在Section 40.3.1和 Section 40.6.1中。

40.2. PL/pgSQL的结构

PL/pgSQL是一种块结构的语言。 函数定义的所有文本都必须是一个块（*block*）。 可以用下面的方法定义一个块：

```
[ <<`_label_`>> ]
[ DECLARE
    `_declarations_` ]
BEGIN
    _statements_
END [ `_label_` ];
```

块中的每个声明和每条语句都是用一个分号终止的， 如果一个子块在另外一个块里，那么 `END` 后面必须有个分号，如上所述； 不过结束函数体的最后的 `END` 可以不要这个分号。

> **Tip:** 一个常见的错误是紧跟在 `BEGIN` 之后使用一个分号， 这是不正确的，并且会返回一个语法错误。

如果你想标记出在 `EXIT` 声明中的block，或者描述在block中所声明的变量名字， 此时，可以选择使用 `_label_` 。如果是在 `END` 之后给出一个标签，那么， 它必须与block开始时定义的标签相匹配。

所有的关键字都是不区分大小写的，正如在SQL命令中一样， 会隐式的将其转换成小写，除非是使用双引号。

如同在普通的SQL语句中一样，在PL/pgSQL代码中，用同样的方式定义注释： 在语句的最后，通过一个双破折号(`--`)来开始一条行注释。 而块注释是成对出现的， 通过 `/*` 和 `*/` 来定义。

块语句段里的任何语句都可以是一个子块。子块可以用于逻辑 分组或者把变量局部化为作用于一个较小的语句组。为了子块的持续时间任何同样命名的外部子块的变量在子块中声明变量， 但是如果你符合它们的名字和它们子块标签，无论如何你可以访问外部变量，比如：

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN

RAISE NOTICE 'Quantity here is %', quantity;  -- 在这里的数量是30
    quantity := 50;
    --
-- 创建一个子块
    --
    DECLARE
        quantity integer := 80;
    BEGIN

RAISE NOTICE 'Quantity here is %', quantity;  -- 在这里的数量是80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- 在这里的数量是50
    END;

RAISE NOTICE 'Quantity here is %', quantity;  -- 在这里的数量是50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **Note:** 在任何PL/pgSQL函数主体周围有隐藏的"外部块"。 这个块提供了函数参数的声明（如果有），以及一些特殊变量比如 `FOUND` (参阅Section 40.5.5)。 外部块可以使用函数的名字标记，意味着参数和特殊变量可以满足函数名字。

一定不要把PL/pgSQL里用于语句分组的 `BEGIN` / `END` 和用于事务控制的数据库命令搞混了。PL/pgSQL的 `BEGIN` / `END` 只是用于分组；它们不会开始和结束一个事务。 函数和触发器过程总是在一个由外层命令建立起来的事务里执行， 它们无法开始或者提交事务，因为PostgreSQL没有嵌套事务。 不过，一个包含 `EXCEPTION` 子句的块实际上形成一个子事务，它可以在不影响外层事务的情况下回滚。更多相关信息请参阅Section 40.6.6。

40.3. 声明

所有在块里使用的变量都必须在一个块的声明段里声明。 唯一的例外是一个 `FOR` 循环里的循环变量是在一个整数范围内迭代的， 被自动声明为整数变量。并且同样从游标结果中 `FOR` 循环迭代的循环变量自动被声明为记录变量。

PL/pgSQL变量可以使用任意的SQL数据类型，比如 `integer` , `varchar` 和 `char` 等等。

下面是一些变量声明的例子：

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

一个变量声明的一般性语法是:

```
_name_ [ CONSTANT ] _type_ [ COLLATE `_collation_name_` ] [ NOT NULL ] [ { DEFAULT | := }
```

如果给出了 `DEFAULT` 子句，那么它声明了在进入该块的时候赋予该变量的初始值。 如果没有给出 `DEFAULT` 子句，那么该变量初始化为SQL NULL。 `CONSTANT` 选项避免了该变量被赋值，这样其数值在该块的范围内保持常量。 `COLLATE` 选项声明变量使用的排序规则（参见Section 40.3.6）。 如果声明了 `NOT NULL` ，那么赋予NULL的数值将运行时导致错误。 所以所有声明为 `NOT NULL` 的变量还必须声明一个非空的缺省值。

缺省值是在每次进入该块的时候计算的，而不是每次调用函数时。 因此，如果把 `now()` 赋予一个类型为 `timestamp` 的变量会令变量拥有函数实际调用的时间， 而不是函数预编译的时间。

例如：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

40.3.1. 声明函数参数

传递给函数的参数都是用 `$1` , `$2` 等等这样的标识符。 为了增加可读性，可以为 `$``_n_` 参数名声明别名。 然后别名或者数字标识符都可以指向参数值。

有两种创建别名的方法，比较好的是在 `CREATE FUNCTION` 命令里给出参数名，比如：

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

另外一个方法（也是PostgreSQL 8.0以前的唯一的方法）， 是使用声明语法明确声明别名：

```
_name_ ALIAS FOR $_n_;
```

这个风格的同一个例子看起来像下面这样：

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Note:** 这两个例子的作用不是完全一致的。 在第一个例子中， `subtotal` 可以作为 `sales_tax.subtotal` 被引用， 而在第二个例子中是不可以的。（我们在内部块中附加标签， 反而 `subtotal` 符合这个标签）。

更多例子：

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN    这里放一些使用 v_string 和 index 的计算
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

如果一个PL/pgSQL函数声明中含有输出参数， 那么就会给予输出参数 `$``_n_` 的名字以及可选的别名， 方法和其它正常输入参数一样。一个输出参数实际上是初始值为 NULL 的变量；在函数执行的过程中，应该给它赋值。 该参数的最后数值是返回的东西。比如， 销售额-税费的例子也可以这么做:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

请注意忽略了 `RETURNS real` —当然也可以包含它，不过这样就显得多余了。

输出参数在返回多个数值的时候非常有用。一个简单的例子是：

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

正如在Section 35.4.4里面讨论的， 这样做实际上为函数的结果创建了一个匿名的记录类型。如果给出一个 `RETURNS` 子句，那么它就必须使用 `RETURNS record` 。

另一个声明PL/pgSQL函数的方法是使用 `RETURNS TABLE` ，例如：

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price FROM sales
                 WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

这完全等价于声明一个或多个 `OUT` 参数， 并且声明 `RETURNS SETOF``_sometype_` 。

如果将PL/pgSQL函数的返回类型声明为多态类型 (`anyelement` , `anyarray` , `anynonarray` , `anyenum` , 或者 `anyrange`), 那么就会创建一个特殊的 `$0` 参数， 它的数据类型是函数的实际返回类型， 和从实际输入类型的推导类型一样 (参阅Section 35.2.5)。这样就允许函数像Section 40.3.3里显示的那样访问它的实际返回类型。 `$0` 初始化为空，并且可以被函数修改， 所以，如果需要，它可以用于保存返回值，虽然这并非必须。 `$0` 还可以给予一个别名。 比如，这个函数可以在任何有 `+` 操作符的数据类型上运转：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

通过将一个或多个输出参数声明为多态类型，可以达到相同的效果。 在这种情况下，特殊的参数 `$0` 不会使用；输出参数自己起这个作用。比如：

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

40.3.2. 别名

```
_newname_ ALIAS FOR _oldname_;
```

`别名` 语法比在之前章节提到的更普遍：可以为任何一个参数声明别名，而不仅仅只是对函数。 这样做的主要目的是为已经有名字的参数重新定义一个名字，例如触发器中的 `NEW` 或者 `OLD` 。

例如：

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

由于 `ALIAS` 创建了两种不同的方式来命名相同的对象，因此，无限制的使用会造成混淆。 最好是在重写预定名称时使用。

40.3.3. 拷贝类型

```
_variable_%TYPE
```

`%TYPE` 提供一个变量或者表字段的数据类型。 你可以用这个声明将要保存数据库数值的变量。比如，假如你 在 `users` 表里面有一个 `user_id` 字段。 要声明一个和 `users.user_id` 类型相同的变量，可以这样写：

```
user_id users.user_id%TYPE;
```

通过使用 `%TYPE` ，你无需知道引用的结构的数据类型，并且，最重要的是， 如果被引用项的数据类型在将来变化了(比如把 `user_id` 的类型从 `integer` 改成 `real`)，也不需要修改函数定义。

`%TYPE` 对多态函数特别有用，因为内部变量的数据类型可能在不同调用中不一样。 可以通过给函数的参数或者结果占位符附加 `%TYPE` 的方法来创建合适的变量。

40.3.4. 行类型

```
_name_ _table_name_%ROWTYPE;
_name_ _composite_type_name_;
```

一个复合类型变量叫做行变量(或者*row-type*变量)。 这样的一个变量可以保存一次 `SELECT` 或者 `FOR` 命令结果的完整一行， 只要命令的字段集匹配该变量声明的类型。 行数值的字段使用点表示法访问，比如 `rowvar.field` 。

行变量可以声明为和一个现有的表或者视图的行类型相同， 方法是使用 `_table_name_``%ROWTYPE` 表示法 ; 或者你也可以声明它的类型是一个复合类型的名字。 因为每个表都有一个相关联的同名数据类型，在PostgreSQL里实在是无所谓你写不写 `%ROWTYPE` 。但是有 `%ROWTYPE` 的形式移植性更好。

函数的参数可以是复合类型(表的完整行)。 这个时候，对应的标识符 `$``_n_` 将是一个行变量， 并且可以从中选取字段，比如 `$1.user_id` 。

在一个行类型的变量中，只可以访问用户定义的表中行的属性，不包括OID 或者其它系统属性(因为该行可能来自一个视图)。 该行类型的数据域继承表中像 `char(``_n_`) 这种类型字段的尺寸和精度。

这里是一个使用复合类型的例子。 `table1` 和 `table2` 是现有的表，至少包含代码中提到的字段 :

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

40.3.5. 记录类型

```
_name_ RECORD;
```

纪录变量类似行类型变量，但是它们没有预定义的结构。它们在 `SELECT` 或者 `FOR` 命令中获取实际的行结构。 一个行变量的子结构可以在每次赋值的时候改变。 这样做的一个结果是 : 在一个记录变量被赋予数值之前，它没有子结构， 并且任何对其中的数据域进行访问的企图都将产生一个运行时错误。

请注意， `RECORD` 不是真正的数据类型，只是一个占位符。 还应该意识到在把一个PL/pgSQL 函数声明为返回 `record` 类型的时候，它和一个记录变量的概念并不完全相同， 即使这个函数可能使用一个记录变量保存它的结果也如此。 在这两种情况下书写函数的时候，实际的行结构都是未知的， 但是对于返回 `record` 的函数来说， 实际的结构是在调用它的查询被分析的时候决定的，而行变量可以在运行中改变其行结构。

40.3.6. PL/pgSQL变量的排序规则

当PL/pgSQL函数有排序规则数据类型的一个以上的参数时， 排序规则确定每个函数调用依赖于分配给实际参数的排序规则，正如Section 22.2。 如果排序规则成功被识别（比如，在这些参数之间没有隐式排序规则冲突），那么所有 排序规则参数作为有隐式排序规则对待。 这将影响函数内部排序规则敏感操作行为。比如，考虑：

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

`less_than` 的第一次使用出于比较将使用 `text_field_1` 和 `text_field_2` 的通用排序规则，然而第二次使用将使用 `C` 排序规则。

此外，被识别的排序规则也被假定为任何局部变量是collatable类型的排序规则。 因此这个函数没有任何不同，如果它被写为：

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

如果没有collatable数据类型的参数，或者没有通用排序规则可以识别他们，那么参数和局部变量 使用数据类型的缺省排序规则（这往往是数据库的缺省排序规则，但是可能不同于域类型变量）。

collatable数据类型的局部变量可以有与声明中包含 `COLLATE` 选项的相关联的不同排序规则。比如，

```
DECLARE
    local_a text COLLATE "en_US";
```

这个选项覆盖排序规则，否则按照上述规则给定变量。

同时，如果期望强迫在特定操作中使用特定排序规则，当然明确的 `COLLATE` 子句可以写 在函数中。

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

这将重写与表列，参数，或者表达式中使用的局部变量相关联的排序规则，正如在纯SQL命令中一样。

40.4. 表达式

所有在PL/pgSQL语句里使用的表达式都是用服务器的普通 SQL执行器进行处理的。 例如，当要写一个如下的PL/pgSQL声明时

```
IF _expression_ THEN ...
```

PL/pgSQL会通过在SQL引擎中输入类似下面的查询来计算表达式

```
SELECT _expression_
```

一旦形成 `SELECT` 命令，任何出现的PL/pgSQL变量名会由参数取代， 正如在Section 40.10.1讨论的那样。 因此，只需要定义一次SELECT查询计划， 之后可以重复使用。也就是说，第一次使用表达式时， 本质上是生效的是 `PREPARE` 命令。 例如，我们声明两个整型变量 `x` 和 `y` ：

```
IF x < y THEN ...
```

后台实际执行的是：

```
PREPARE _statement_name_(integer, integer) AS SELECT $1 < $2;
```

并且，一条 `EXECUTE` 说明语句会处于预备状态， 以后每一次执行 `IF` 语句时都会调用该说明语句， 将当前PL/pgSQL变量的值提供为参数值。 通常情况下，对于PL/pgSQL用户来说，这样做并不是特别重要，不过， 当在进行错误诊断时，如果知道这一点的话会很有用。 更多详细信息参阅Section 40.10.2。

40.5. 基本语句

本节以及随后的一节里， 描述所有PL/pgSQL明确可以理解的语句类型。 任何无法识别为这样类型的语句将被做为SQL命令看待， 并且被发送到主数据库引擎执行， 正如在节Section 40.5.2 和Section 40.5.3中描述的那样。

40.5.1. 赋值

给一个PL/pgSQL变量的赋值如下:

```
_variable_ := _expression_;
```

如上所述，语句中的表达式是用一个发送到主数据库引擎的 `SELECT` 命令计算的。 该表达式必须生成单一的数值。表达式必须只能生成一个值 （如果变量一个行或者record，那么该值可能是一个行）。 目标变量可以是一个简单的变量（可以用一个block的名字来描述）， 行或record变量的字段，或者是一个简单变量或字段的数组元素。

如果表达式的结果数据类型和变量数据类型不一致， 或者变量具有已知的尺寸/精度(比如 `char(20)`)， 结果值将隐含地被PL/pgSQL解释器用结果类型的输出 函数和变量类型的输入函数转换。 注意，如果结果数值的字符串形式不是输入函数可以接受的形式， 那么这样做可能导致类型输入函数产生的运行时错误。

例子：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

40.5.2. 执行一个没有结果的查询

对于不返回任何行的SQL命令，例如没有 `RETURNING` 子句的 `INSERT` ， 你可以简单的在PL/pgSQL函数内写上该语句， 然后执行该函数即可。

出现在查询文本中的任何PL/pgSQL变量名都会被参数符号代替， 并在运行时将参数值替换为变量的当前值。 就像之前描述的表达式进程，可以查看资料Section 40.10.1。

当以这种方式执行一条SQL命令，这条命令在PL/pgSQL 中缓存并且在执行规划中重新使用。正如在Section 40.10.2中讨论的。

有时评估一个表达式或 `SELECT` 查询但是丢弃其结果也是有用的， 例如，调用一个具有副作用的函数，但对它的结果不感兴趣。 要在PL/pgSQL中这样做，可以使用 `PERFORM` 语句:

```
PERFORM _query_;
```

这将执行 `_query_` 并丢弃其结果。 用 `SELECT` 命令重写 `_query_`，并将 `SELECT` 替换为 `PERFORM`， 对于 `WITH` 查询，使用 `PERFORM` 并且将查询放在括号中（在这种情况下， 查询只仅仅返回一行）。 这样，PL/pgSQL变量将会在查询中被照常替换。 另外，如果查询生成至少一行结果的话， 特殊变量 `FOUND` 将会被设为真，否则将被设为假。 （查阅Section 40.5.5）

> **Note:** 有些人可能期望直接写 `SELECT` 就能同样达到此目的， 但目前确实只有 `PERFORM` 一种方法。 诸如 `SELECT` 这样返回行的查询将会被当作错误拒绝， 除非其带有一个下面将要讨论的 `INTO` 子句。

例如：

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

40.5.3. 执行一个仅有单行结果的查询

如果一个SQL命令的结果是一个单独的行(可能有多个字段)， 那么可以将其赋予一个记录变量、行类型变量、标量变量的列表。 这可以通过在基本SQL命令之后添加一个 `INTO` 子句达到。例如：

```
SELECT _select_expressions_ INTO [STRICT] _target_ FROM ...;
INSERT ... RETURNING _expressions_ INTO [STRICT] _target_;
UPDATE ... RETURNING _expressions_ INTO [STRICT] _target_;
DELETE ... RETURNING _expressions_ INTO [STRICT] _target_;
```

这里的 `_target_` 可以是一个记录变量、 行变量、逗号分隔的简单变量列表、逗号分隔记录/行字段列表。 PL/pgSQL变量将被照常代入查询的其余部分， 适用于带有 `RETURNING` 的 `SELECT`，`INSERT` / `UPDATE` / `DELETE`， 以及返回行集合的命令(比如 `EXPLAIN`)。 除 `INTO` 子句外， SQL命令与其在PL/pgSQL外面时完全相同。

> **Tip:** 请注意，上面带有 `INTO` 的 `SELECT` 和 PostgreSQL普通的 `SELECT INTO` 命令是不一样的， 后者的 `INTO` 目标是一个新创建的表。 如果你想在PL/pgSQL函数里从一个 `SELECT` 结果中创建表， 那么请使用 `CREATE TABLE ... AS SELECT` 语法。

如果将一行或者一个变量列表用做目标，那么查询的结果必需作为数目或者数据类型精确匹配目标的结构， 否则就会产生运行时错误。如果目标是一个记录变量，那么它自动将自己配置成命令结果列的行类型。

`INTO` 子句几乎可以出现在SQL命令的任何地方。 习惯上把它写在 `SELECT` 命令的 `_select_expressions_` 列表的之前或之后， 对于其它命令则位于结尾。 我们建议你遵守这个约定，以防万一PL/pgSQL分析器在未来的版本中变得更加严格。

如果没有在INTO指定STRICT，那么target将被设为查询返回结果的第一行或者 NULL(查询返回零行)， 请注意，除非用ORDER BY进行排序，否则"the first row"是不明确的。 第一行之后的所有结果都将被丢弃。 你可以检查特殊变量FOUND(参见Section 39.5.5)来判断查询是否至少返回一行。

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

如果指定了 `STRICT` 选项， 那么查询必须返回恰好一个行或者是运行时的错误， 要么是 `NO_DATA_FOUND` (没有行)，要么是 `TOO_MANY_ROWS` (多于一行)。 可以使用异常块来捕获这些错误。例如：

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

成功执行了一个带有 `STRICT` 的命令之后， `FOUND` 将总是被设为真。

对于带有 `RETURNING` 的 `INSERT` / `UPDATE` / `DELETE` ， 即使没有指定 `STRICT` ，PL/pgSQL也会在返回多行时报错。 这是因为没有 `ORDER BY` 之类的选项用于确定究竟返回那一行。

> **Note:** `STRICT` 兼容 Oracle PL/SQL的 `SELECT INTO` 行为以及相关语句。

对于如何处理一个SQL查询中返回的多行，参见Section 40.6.4。

40.5.4. 执行动态命令

你经常会希望在你的PL/pgSQL函数里生成动态命令。 也就是那些每次执行的时候都会涉及不同表或不同数据类型的命令。在这样的情况下， PL/pgSQL试图为命令(正如Section 40.10.2讨论的) 缓冲执行计划的一般企图将不再合适。 为了处理这样的问题，提供了 `EXECUTE` 语句：

```
EXECUTE _command-string_ [ INTO [STRICT] `_target_` ] [ USING `_expression_` [, ... ] ];
```

这里的 `_command-string_` 是一个生成字符串(类型为 `text`)的表达式， 该字符串包含要执行的命令。 而 `_target_` 是一个记录变量、行变量、逗号分隔的简单变量列表、 逗号分隔的记录/行列表，来存储命令的结果。通过使用 `USING` 表达式，将参数值插入到命令中。

请特别注意在该命令字符串里将不会发生任何PL/pgSQL变量代换。 变量的数值必需在构造的时候插入该字符串的值，或者也可以使用下面介绍的参数。

同时，对于通过 `EXECUTE` 执行的命令，没有预先设置缓存计划。 相反，在该语句每次运行的时候，命令都准备一次。 命令字符串可以在过程里动态地生成以便于对各种不同的表和字段进行操作。

`INTO` 子句声明SQL命令的结果应该传递到哪里。 如果提供了一个行变量或者一个变量列表， 那么它必须和查询生成的结果的结构一样(如果使用了记录变量，那么它回自动调整为匹配结果的结构)。 如果返回了多行，那么只有第一行将被赋予 `INTO` 变量。 如果返回零行，那么将给 `INTO` 变量赋予NULL。 如果没有声明 `INTO` 子句，则抛弃查询结果。

如果使用了 `STRICT` 选项，那么在查询没有恰好返回一个行的情况下将会报错。

该命令可以使用那些在命令中被引用为 `$1`，`$2` 等的参数值。 这些标签指向的是在 `USING` 子句中使用的值。 这样做可以很好的将数据值以文本类型插入到命令字符串中： 避免了运行期间在数据值和文本类型之间转换的开销， 并且这种方法不是倾向于进行SQL-injection，因为没有进行引用和转义的必要。例如：

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

需要注意的是，参数标签只能用于数据值— 如果想要使用动态的已知的表或列的名字， 那么必须将它们以文本字符串类型插入到命令中。 例如，当上面那个查询需要在一个动态选择的表上执行时，你可以这么做：

```
EXECUTE 'SELECT count(*) FROM '
    || tabname::regclass
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

另一个关于参数标签的限制是， 它们只能在 `SELECT`，`INSERT`，`UPDATE` 和 `DELETE` 命令中使用。 在另一种语法类型中，通常称为通用语法中，可以将参数值以文本类型插入， 哪怕它们只是数据值。

如在上面第一个例子中的，带有一个简单常量字符串和 `USING` 参数的 `EXECUTE` 命令， 它在功能上等同于直接在PL/pgSQL中写命令， 并且允许PL/pgSQL变量自动替换。 最重要的不同之处在于， `EXECUTE` 会在每一次执行时，根据当前的参数值更新该命令计划， 在这一点上，PL/pgSQL可能创建一个命令计划，并将 其放于缓存中以待重新使用。 当命令计划对参数值的依赖性很强时， 对于使用 `EXECUTE` 积极确保通用计划不被选择是很有帮助的。

`EXECUTE` 命令目前不支持 `SELECT INTO`， 但是支持一个纯 `SELECT` 命令，并且声明一个 `INTO` 作为命令本身的一部分。

> **Note:** PL/pgSQL中的 `EXECUTE` 语法与 PostgreSQL服务器支持的EXECUTE语法无关。 服务器支持的 `EXECUTE` 语法不能 被PL/pgSQL函数直接使用（并且也没有必要）。

Example 40-1. 动态查询中的引用值

使用动态命令的时候经常需要逃逸单引号。 建议使用美元符界定函数体内的固定文本。 如果你有没有使用美元符界定的老代码，请参考Section 40.11.1， 这样在把老代码转换成更合理的结构时，会节省你的一些精力。

插入到构造出来的查询中的动态数值也需要特殊处理， 因为他们自己可能包含引号字符。 一个例子(这里都假设你使用了美元符作为整体，所以引号标记不需要加倍)：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

这个例子显示了 `quote_ident` 和 `quote_literal` 函数的使用（参阅Section 9.4）。 为了安全，包含字段和表标识符的变量应该传递给 `quote_ident` 函数。 那些包含数值的表达式，如果中的数值在构造出来的命令字符串里是文本字符串， 那么应该传递给 `quote_literal` 。 它们俩都会采取合适的步骤把输入文本包围在单或双引号里， 并且对任何嵌入其中的特殊字符进行合适的逃逸处理。

因为 `quote_literal` 被标记为 `STRICT` ， 当发出带有null参数的请求时， 往往会返回一个null。在上面的例子中，如果 `newvalue` 或者 `keyvalue` 是null， 整个动态查询字符串会变成null，最终 `EXECUTE` 会报错。 可以通过使用 `quote_nullable` 函数来避免该错误， 除了当发出带有null参数的请求时，往往会返回一个字符串NULL之外， 该函数与 `quote_literal` 一样工作。例如：

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

如果处理的参数值是null，那么应该用 `quote_nullable` 来代替 `quote_literal` 。

通常， 应该注意确保查询中的null值返回意料之外的结果。例如：

```
'WHERE key = ' || quote_nullable(keyvalue)
```

如果 `keyvalue` 是null，那么该 `WHERE` 子句永远不会成功， 因为当 `=` 操作符带有null操作数，操作返回的结果往往是null。 如果想让null同普通关键字一样使用，那么将上面的命令修改如下：

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

目前，`IS NOT DISTINCT FROM` 处理效率不如 `=`，因此如非必要，不用这么做。关于null和 `IS DISTINCT` 的资料可参阅Section 9.2。

请注意美元符界定只对包围固定文本有用。如果想像下面这样做上面的例子，那就太糟糕了:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

因为如果 `newvalue` 的内容碰巧含有 `$$`，那么这段代码就有毛病了。同样的问题可能出现在你选用的任何美元符界定分隔符上。因此，要想安全地包围事先不知道的文本，必须恰当的使用 `quote_literal`，`quote_nullable` 或者 `quote_ident`。

动态SQL语句可以使用 `format` 函数安全构建（参阅Section 9.4）。比如：

```
EXECUTE format('UPDATE tbl SET %I = %L WHERE key = %L', colname, newvalue, keyvalue);
```

在 `USING` 子句连接中使用 `format` 函数：

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

这种形式更有效，因为参数 `newvalue` and `keyvalue` 不转换为文本。

关于动态命令和 `EXECUTE` 的另一个例子 是Example 40-9，这个例子制作并执行了一个定义新函数的 `CREATE FUNCTION` 命令。

40.5.5. 获取结果状态

有好几种方法可以判断一条命令的效果。第一个方法是使用 `GET DIAGNOSTICS`，它的形式如下：

```
GET [ CURRENT ] DIAGNOSTICS _variable_ = _item_ [ , ... ];
```

这条命令允许检索系统状态标识符。每个 `_item_` 是一个关键字，表示一个将要赋予该特定变量的状态值(该变量应该和要接收的数值类型相同)。当前可用的状态项有 `ROW_COUNT`、最后一个SQL命 令发送到SQL引擎处理的行数量、`RESULT_OID`，最后一条SQL命令插入的最后一行的OID。请注意 `RESULT_OID` 只有在一个向包含OID的表中 `INSERT` 的命令之后才有用。

例如：

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

另外一个判断命令效果的方法是一个 `boolean` 类型的特殊变量 `FOUND` ， 它在每个PL/pgSQL函数调用中 `FOUND` 开始都为假。 并被下列语句设置：

- 一个 `SELECT INTO` 语句如果返回一行则将 `FOUND` 设置为真， 如果没有返回行则设置为假。
- 一个 `PERFORM` 语句如果生成(或抛弃)一行，则将 `FOUND` 设置为真， 如果没有生成行则为假。
- 如果至少影响了一行，那么 `UPDATE` , `INSERT` 和 `DELETE` 语句 设置FOUND为真，如果没有行受影响则为假。
- 一个 `FETCH` 语句如果返回行则设置 `FOUND` 为真，如果不返回行则为假
- 当成功定位游标的位置时， `MOVE` 将 `FOUND` 设为真，反之为假。
- 一个 `FOR` 或者 `FOREACH` 语句如果迭代了一次或多次， 则设置 `FOUND` 真，否则为假。 只有在循环退出的时候才设置 `FOUND` ；在循环执行的内部， `FOUND` 不被循环语句修改， 但是在循环体里它可能被其它语句的执行而修改。
- 如果查询结果返回至少一个行， `RETURN QUERY` and `RETURN QUERY EXECUTE` 声明将 `FOUND` 设为真， 反之如果没有返回行，则为假。

其他的PL/pgSQL声明不会改变 `FOUND` 的位置。 尤其需要注意的一点是： `EXECUTE` 会修改 `GET DIAGNOSTICS` 的输出， 但不会修改 `FOUND` 的输出。

`FOUND` 是每个PL/pgSQL里的局部变量；任何对它的任何修改只影响当前的函数。

40.5.6. 什么也不做

有时一个什么也不做的占位语句也是很有用的。 例如，用于if/then/else 的空分支。 可以使用 `NULL` 语句达到这个目的。

```
NULL;
```

比如，下面的两段代码是相等的：

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignore the error
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ignore the error
END;
```

究竟使用哪一个取决于个人的喜好。

> **Note:** 在Oracle的PL/SQL中，不允许出现空语句列， 所以在这种情况下必须使用`NULL`语句， 而PL/pgSQL允许你什么也不写。

40.6. 控制结构

控制结构可能是PL/pgSQL中最有用的(以及最重要)的部分了。 利用PL/pgSQL的控制结构，你可以以非常灵活而且强大的方法操纵PostgreSQL的数据。

40.6.1. 从函数返回

有两个命令可以用来从函数中返回数据：`RETURN` 和 `RETURN NEXT` 。

40.6.1.1. `RETURN`

```
RETURN _expression_;
```

带表达式的 `RETURN` 用于终止函数 并把 `_expression_` 的值返回给调用者。 这种形式用于不返回集合的PL/pgSQL函数。

如果函数中返回标量类型，那么表达式结果将被自动转换成函数的返回类型， 就像在赋值中描述的那样。但是要返回一个复合(行)数值， 你必须写一个准确提供需求列集合的表达式，这可能需要显式转换。

如果你声明带有输出参数的函数，那么就只需要写无表达式的 `RETURN` 。 那么输出参数变量的当前值将被返回。

如果你声明函数返回 `void` ，那么一个 `RETURN` 语句可以用于提前退出函数； 但是不要在 `RETURN` 后面写一个表达式。

一个函数的返回值不能是未定义。 如果控制到达了函数最顶层的块而没有碰到一个 `RETURN` 语句， 那么它就会发生一个错误。不过，这个限制不适用于带输出参数的函数以及那些返回 `void` 的函数。 在这些例子里，如果顶层的块结束，则自动执行一个 `RETURN` 语句。

例子：

```
-- 返回一个标量类型函数
RETURN 1 + 2;
RETURN scalar_var;
-- 返回复合类型函数
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct types
```

40.6.1.2. `RETURN NEXT` 和 `RETURN QUERY`

```
RETURN NEXT _expression_;
RETURN QUERY _query_;
RETURN QUERY EXECUTE _command-string_ [ USING `_expression_` [, ... ] ];
```

如果一个PL/pgSQL函数声明为返回 `SETOF` `_sometype_`， 那么遵循的过程则略有不同。 在这种情况下，要返回的独立项是在 `RETURN NEXT` 或者 `RETURN QUERY` 命令里声明的， 然后最后有一个不带参数的 `RETURN` 命令用于告诉这个函数已经完成执行了。 `RETURN NEXT` 可以用于标量和复合数据类型；对于复合类型， 将返回一个完整的结果"table"。 `RETURN QUERY` 命令将一条查询的结果追加到一个函数的结果集中。 `RETURN NEXT` 和 `RETURN QUERY` 在单一集合返回 函数中自由混合，在这种情况下，结果将被级联。

`RETURN NEXT` 和 `RETURN QUERY` 实际上不会从函数中返回， 它们是将零或者多个行追加到函数的结果集中。 然后继续执行PL/pgSQL函数里的下一条语句。 随着后继的 `RETURN NEXT` 或者 `RETURN QUERY` 命令的执行， 结果集就建立起来了。最后一个 `RETURN` 应该没有参数， 它导致控制退出该函数(或者你可以简单地让控制到达函数的结尾)。

`RETURN QUERY` 有一个变形 `RETURN QUERY EXECUTE`， 指定查询将被动态执行。 参数表达式可以通过 `USING` 插入到计算查询字符串中，以 `EXECUTE` 命令的同样方式。

如果你声明函数带有输出参数，那么就只需要写不带表达式的 `RETURN NEXT`。 输出参数的当前值将被保存，用于最终返回。请注意如果有多个输出参数， 比如声明函数为返回 `SETOF record` 或者是在只有一个类 型为 `_sometype_` 的输出参数时声明 为 `SETOF _sometype_`， 这样才能创建一个带有输出参数的返回集合的函数。

下面是一个使用 `RETURN NEXT` 的函数例子:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP   -- 可以在这里做一些处理
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

这是一个使用 `RETURN QUERY` 的函数例子:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);
    -- 由于没有完成执行，我们可以检查行是否返回并且如果没有则抛出异常。
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;
-- 如果没有可用航班，则返回可用航班或者抛出异常。
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> **Note:** 目前 `RETURN NEXT` 和 `RETURN QUERY` 实现在从函数返回之前把整个结果集都保存起来， 就像上面描述的那样。这意味着如果一个PL/pgSQL函数生成一个非常大的结果集， 性能可能会很差：数据将被写到磁盘上以避免内存耗尽， 但是函数在完成整个结果集的生成之前不会退出。 将来的PL/pgSQL版本可能会允许用户定义没有这样限制的返回集合的函数。 目前，数据开始向磁盘里写的时刻是由配置变量work_mem控制的。 拥有足够内存的管理员如果想在内存里存储更大的结果集， 则可以考虑把这个参数增大一些。

40.6.2. 条件

`IF` 和 `CASE` 语句让你可以根据某种条件执行命令。 PL/pgSQL有三种形式的 `IF` ：

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

以及两种形式的 `CASE` ：

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

40.6.2.1. `IF-THEN`

```
IF _boolean-expression_ THEN
    _statements_
END IF;
```

`IF-THEN` 语句是 `IF` 的最简单形式。如果条件为真， 在 `THEN` 和 `END IF` 之间的语句将被执行。 否则，将忽略它们。

例如：

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

40.6.2.2. `IF-THEN-ELSE`

```
IF _boolean-expression_ THEN
    _statements_
ELSE
    _statements_
END IF;
```

`IF-THEN-ELSE` 语句增加了 `IF-THEN` 的分支， 让你可以声明在条件为假的时候执行的语句。（请注意这包含条件是NULL的情况）。

例如：

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

40.6.2.3. `IF-THEN-ELSIF`

```
IF _boolean-expression_ THEN
    _statements_
[ ELSIF `_boolean-expression_` THEN
    `_statements_`
[ ELSIF `_boolean-expression_` THEN
    `_statements_`
    ...]]
[ ELSE
    `_statements_` ]
END IF;
```

有时不止两个选择。 `IF-THEN-ELSIF` 反过来提供了一个简便的方法来检查选择条件。 `IF` 判断会陆续检查，直到找到第一个为真的，然后执行相关声明，如此， 直到 `END IF` （不会检测 `IF` 子查询）。 如果没有一个条件符合 `IF` 判断，那么会接着执行 `ELSE` 判断。

例如：

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE    -- 唯一可能性是号码为空
    result := 'NULL';
END IF;
```

`ELSIF` 关键字也可以写成 `ELSEIF` 。

另一个可以实现该目的的方法是使用 `IF-THEN-ELSE` 声明，如下:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

然而，这个方法需要为每个 `IF` 写 `END IF` ， 因此当有很多选择时，这种方法明显比 `ELSIF` 繁琐。

40.6.2.4. 简单 `CASE`

```
CASE _search-expression_
    WHEN _expression_ [, `_expression_` [ ... ]] THEN
      _statements_
  [ WHEN `_expression_` [, `_expression_` [ ... ]] THEN
      `_statements_`
    ... ]
  [ ELSE
      `_statements_` ]
END CASE;
```

`CASE` 简单的形式提供基于操作数平等的条件执行。 `_search-expression_` 被评价并且 先后比较 `WHEN` 子句中的每个 `_表达式_` 。 如果找到匹配，那么相应的 `_statements_` 被执行， 然后控制在 `END CASE` 之后传递到下一个语句。 （随后的 `WHEN` 表达式不被评估。） 如果没有发现匹配，执行 `ELSE` `_statements_` ；但如果 `ELSE` 是不存在的，然后引发 `CASE_NOT_FOUND` 异常。

例如：

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

40.6.2.5. 搜索 `CASE`

```
CASE
    WHEN _boolean-expression_ THEN
      _statements_
  [ WHEN `_boolean-expression_` THEN
      `_statements_`
    ... ]
  [ ELSE
      `_statements_` ]
END CASE;
```

`CASE` 搜索形式基于布尔表达式的真理提供条件执行。 每个 `WHEN` 子句的 `_boolean-expression_` 依次被评估， 直到找到一个产生 `true` 为止。 然后执行相应的 `_statements_` ， 控制 `END CASE` 之后传递到下一个语句。 （随后不评估 `WHEN` 表达式）。 如果发现没有真实结果，则执行 `ELSE``_statements_` ；但如果 `ELSE` 是不存在的，那么引发 `CASE_NOT_FOUND` 异常。

例如：

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

`CASE` 这种形式完全等价于 `IF-THEN-ELSIF` ，除了达到忽略错误中的 `ELSE` 子句结果而不是什么都不做的规则。

40.6.3. 简单循环

使用 `LOOP` , `EXIT` , `CONTINUE` , `WHILE` ， `FOR` 和 `FOREACH` 语句， 可以控制PL/pgSQL函数重复一系列命令。

40.6.3.1. `循环`

```
[ <<`_label_`>> ]
LOOP
    _statements_
END LOOP [ `_label_` ];
```

`LOOP` 定义一个无条件的循环，无限循环， 直到由 `EXIT` 或者 `RETURN` 语句终止。 可选的 `_label_` 可以由 `EXIT` 和 `CONTINUE` 语句使用， 用于在嵌套循环中声明应该应用于哪一层循环。

40.6.3.2. `退出`

```
EXIT [ `_label_` ] [ WHEN `_boolean-expression_` ];
```

如果没有给出 `_label_`， 那么退出最内层的循环， 然后执行跟在 `END LOOP` 后面的语句。 如果给出 `_label_`， 那么它必须是当前或者更高层的嵌套循环块或者语句块的标签。 然后该命名块或者循环就会终止，而控制落到对应循环/块的 `END` 语句后面的语句上。

如果声明了 `WHEN`， 循环退出只有在 `_boolean-expression_` 为真的时候才发生， 否则控制会落到 `EXIT` 后面的语句上。

`EXIT` 可以用于在所有的循环类型中，它并不仅仅限制于在无条件循环中使用。

在和 `BEGIN` 块一起使用的时候， `EXIT` 把控制交给块结束后的下一个语句。 需要注意的是，一个标签必须用于这个目的； 一个没有标记的 `EXIT` 永远无法与 `BEGIN` 进行匹配。 （这是PostgreSQL 8.4之前版本的一个变化， 这将允许未标记 `EXIT` 匹配 `BEGIN` 块）。

例如：

```
LOOP

-- 一些计算
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP

-- 一些计算
EXIT WHEN count > 0;  -- 和前面的例子相同结果
END LOOP;

<<ablock>>
BEGIN

-- 一些计算
    IF stocks > 100000 THEN
    EXIT ablock;  -- 导致从BEGIN块退出
    END IF;
 -- 忽略这儿的计算，当stocks > 100000时
END;
```

40.6.3.3. `CONTINUE`

```
CONTINUE [ `_label_` ] [ WHEN `_boolean-expression_` ];
```

如果没有给出 `_label_`，那么就开始最内层循环的下一次执行。 也就是说，控制传递回给循环控制表达式(如果有)，然后重新计算循环体。 如果出现了 `_label_`，它声明即将继续执行的循环的标签。

如果声明了 `WHEN`，那么循环的下一次执行只有 在 `_boolean-expression_` 为真的情况下才进行。 否则，控制传递给 `CONTINUE` 后面的语句。

`CONTINUE` 可以用于所有类型的循环；它并不仅仅限于无条件循环。

例如：

```
LOOP

-- 一些计算
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
-- 在[50 .. 100]内的计算
END LOOP;
```

40.6.3.4. `WHILE`

```
[ <<`_label_`>> ]
WHILE _boolean-expression_ LOOP
    _statements_
END LOOP [ `_label_` ];
```

只要条件表达式（ `_boolean-expression_` ）为真， `WHILE` 语句就会不停的在一系列语句上进行循环， 条件是在每次进入循环体的时候被检查。

例如：

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- 这里的一些计算
END LOOP;

WHILE NOT done LOOP
    -- 这里的一些计算
END LOOP;
```

40.6.3.5. `FOR` (Integer 变量)

```
[ <<`_label_`>> ]
FOR _name_ IN [ REVERSE ] _expression_ .. _expression_ [ BY `_expression_` ] LOOP
    _statements_
END LOOP [ `_label_` ];
```

这种形式的 `FOR` 对一定范围的整数进行迭代的循环。 变量 `_name_` 会自动定义为 `BY` 类型并且只在循环里存在 (任何该变量名的现存定义在此循环内都将被忽略)。 给出范围上下界的两个表达式在进入循环的时候计算一次。 `BY` 子句指定迭代步长(缺省为 1)， 但如果声明

了 `REVERSE` 步长将变为相应的负值。

一些整数 `FOR` 循环的例子：

```
FOR i IN 1..10 LOOP
    -- 我将在值1,2,3,4,5,6,7,8,9,10中循环
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- 将在值10,9,8,7,6,5,4,3,2,1中循环
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- 将在值10,8,6,4,2中循环
END LOOP;
```

如果下界大于上界(或者是在 `REVERSE` 情况下是小于)， 那么循环体将完全不被执行。 而且不会抛出任何错误。

如果 `_label_` 被附加到 `FOR` 循环，那么整数循环变量 可以使用 `_label_` 引用适当名称。

40.6.4. 遍历命令结果

使用不同类型的 `FOR` 循环， 你可以遍历一个命令的结果并且对其进行相应的操作。语法是：

```
[ <<`_label_`>> ]
FOR _target_ IN _query_ LOOP
    _statements_
END LOOP [ `_label_` ];
```

`_target_` 是一个记录变量、 行变量、逗号分隔的标量变量列表 `_target_` 被连续不断赋予所有来自 `_query_` 的行， 并且循环体将为每行执行一次。 下面是一个例子：

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

-- 现在"mviews"里有了一条来自 cs_materialized_views 的记录

        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO '
                   || quote_ident(mviews.mv_name) || ' '
                   || mviews.mv_query;
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

如果循环是用一个 `EXIT` 语句终止的， 那么在循环之后你仍然可以访问最后赋值的行。

`FOR` 语句中使用的这种 `_query_` 可以是任何返回行的SQL命令， 通常是 `SELECT` ， 不过带有 `RETURNING` 子句的 `INSERT` , `UPDATE` 或 `DELETE` 也是可以的， 一些诸如 `EXPLAIN` 之类的命令也可以。

PL/pgSQL变量代替查询文本，并且查询计划为了重新使用被缓存，正如 Section 40.10.1和 Section 40.10.2。

`FOR-IN-EXECUTE` 语句是遍历所有行的另外一种方法：

```
[ <<`_label_`>> ]
FOR _target_ IN EXECUTE _text_expression_ [ USING `_expression_` [, ... ] ] LOOP
    _statements_
END LOOP [ `_label_` ];
```

这个例子类似前面的形式，只不过源查询语句声明为了一个字符串表达式， 这样它在每次进入 `FOR` 循环的时候都会重新计算和生成执行计划。 这样就允许程序员在一个预先规划好了的命令所获得的速度和一个动态命令所获得的灵活性 (就像一个简单的 `EXECUTE` 语句那样)之间进行选择。 当使用 `EXECUTE` 时， 可以通过 `USING` 将参数值插入到动态命令中。

对于一个需要将结果迭代的查询， 另外一个声明的方法是将它定义为游标（cursor）， 可参阅Section 40.7.4。

40.6.5. 遍历数组

`FOREACH` 循环类似于 `FOR` 循环， 但不是遍历SQL查询返回的行，它遍历数组值元素。 （一般而言， `FOREACH` 是遍历复合值表达式组成部分； 循环遍历除数组外的复合值变量将来可以被添加。） `FOREACH` 语句循环数组是：

```
[ <<`_label_`>> ]
FOREACH _target_ [ SLICE `_number_` ] IN ARRAY _expression_ LOOP
    _statements_
END LOOP [ `_label_` ];
```

没有 `SLICE` ， 或者如果声明 `SLICE 0` ， 则 循环遍历通过评估 `_expression_` 产生的数组的单个元素。 `_target_` 变量分配每个 序列中的元素值，并为每个元素执行循环体。 这里是遍历整数数组元素的一个例子：

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

元素以存储顺序进行访问，不论数组维数的数量。尽管 `_target_` 通常只是一个单一的变量，当循环复合值的数组（记录）时，它可以是一个变量列表， 在这种情况下，每个数组元素，从连续的复合值列中分配变量。

以正数 `SLICE` 值， `FOREACH` 遍历数组的元素部分，而不是单一元素。 `SLICE` 的值必须是不大于数组维数的整数常数。 `_target_` 变量必须是一个数组， 并且它接收数组值的连续片段，而每个片段 是通过 `SLICE` 指定的维数。这里是遍历一维切片的一个例子：

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

40.6.6. 捕获错误

缺省时，一个在PL/pgSQL函数里发生的错误退出函数的执行， 并且实际上其周围的事务也会退出。 你可以使用一个带有 `EXCEPTION` 子句的 `BEGIN` 块捕获错误并且从中恢复。 其语法是正常的 `BEGIN` 块语法的一个扩展：

```
[ <<`_label_`>> ]
[ DECLARE
    `_declarations_` ]
BEGIN
    _statements_
EXCEPTION
    WHEN _condition_ [ OR `_condition_` ... ] THEN
        _handler_statements_
    [ WHEN `_condition_` [ OR `_condition_` ... ] THEN
          `_handler_statements_`
      ... ]
END;
```

如果没有发生错误，这种形式的块只是简单地执行所有 `_statements_` ， 然后转到下一个 `END` 之后的语句。 但是如果在 `_statements_` 内部发生了一个错误， 则对 `_statements_` 的进一步处理将废弃，然后转到 `EXCEPTION` 列表。 系统搜索这个列表，寻找匹配错误的第一个 `_condition_` 。 如果找到匹配，则执行对应的 `_handler_statements_` ， 然后转到 `END` 之后的下一个语句。如果没有找到匹配，该错误就会广播出去， 就好像根本没有 `EXCEPTION` 子句一样： 该错误可以被一个包围块用 `EXCEPTION` 捕获， 如果没有包围块，则退出函数的处理。

`_condition_` 的名字可以是Appendix A里显示的任何名字。 一个范畴名匹配任意该范畴里的错误。 特殊的条件名 `OTHERS` 匹配除了 `QUERY_CANCELED` 之外的所有错误类型。 可以用名字捕获 `QUERY_CANCELED` ，不过通常是不明智的。 条件名是大小写无关的。同时也可以通过 `SQLSTATE` 来声明一个错误条件， 例如：

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

如果在选中的 `_handler_statements_` 里发生了新错误， 那么它不能被这个 `EXCEPTION` 子句捕获，而是传播出去。 一个外层的 `EXCEPTION` 子句可以捕获它。

如果一个错误被 `EXCEPTION` 捕获，PL/pgSQL函数的局部变量保持错误发生时的原值， 但是所有该块中想固化在数据库中的状态都回滚。 作为一个例子，让我们看看下面片断:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

当控制到达给 `y` 赋值的地方时， 它会带着一个 `division_by_zero` 错误失败。 这个错误将被 `EXCEPTION` 子句捕获。 而在 `RETURN` 语句里返回的数值将是 `x` 的增量值。 但是 `UPDATE` 已经被回滚。然而，在该块之前的 `INSERT` 将不会回滚， 因此最终的结果是数据库包含 `Tom Jones` 而不是 `Joe Jones` 。

> **Tip:** 进入和退出一个包含 `EXCEPTION` 子句的块要比不包含的块开销大的多。 因此，不必要的时候不要使用 `EXCEPTION` 。

Example 40-2. `UPDATE` / `INSERT` 异常

这个例子根据使用异常处理器执行恰当的 `UPDATE` 或者 `INSERT` 。

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP

-- 第一次尝试更新key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
-- 不存在，所以尝试插入key,如果其他人同时插入相同的key，我们可能得到唯一key失败。
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
    -- 什么也不做，并且循环尝试再次更新。
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

这个代码假设通过 `INSERT` 不是说表上触发器函数中的 `INSERT` 产生 `unique_violation` 错误，如果表上有超过一个以上的唯一索引，它可能行为不端， 因为将重试操作不论哪个索引产生错误。 可以通过特性讨论下一步检查捕获的错误是预期的来获取更高的安全性。

40.6.6.1. 获得有关错误的信息

异常处理程序经常需要确定发生的具体错误。有两种方法来获得 当前PL/pgSQL异常： 特殊变量和 `GET STACKED DIAGNOSTICS` 命令 的有关信息。

在一个异常处理程序中，特殊变量 `SQLSTATE` 包含相当于发生异常的错误代码 （参考Table A-1获得可能错误代码列）。 特殊变量 `SQLERRM` 包含与异常有关的错误消息。 这些变量是在异常处理外未被定义的。

在一个异常处理程序中，也可以检索关于使用 `GET STACKED DIAGNOSTICS` 命令的当前异常信息，形成了：

```
GET STACKED DIAGNOSTICS _variable_ = _item_ [ , ... ];
```

每个 `_item_` 是识别被分配到指定变量的状态值（应该是接收它的正确数据类型）的一个关键字。 目前可用的状态显示在Table 40-1中。

Table 40-1. 错误诊断值

名字	类型	描述
`RETURNED_SQLSTATE`	text	异常的SQLSTATE错误代码
`COLUMN_NAME`	text	与异常相关的列名
`CONSTRAINT_NAME`	text	与异常相关的约束名
`PG_DATATYPE_NAME`	text	与异常相关的数据类型名
`MESSAGE_TEXT`	text	异常的主要消息文本
`TABLE_NAME`	text	与异常相关的表名
`SCHEMA_NAME`	text	与异常相关的模式名
`PG_EXCEPTION_DETAIL`	text	异常的详细信息文本，如果任何
`PG_EXCEPTION_HINT`	text	异常的提示信息文本，如果任何
`PG_EXCEPTION_CONTEXT`	text	描述调用堆栈的文本线程

如果异常没有设置项值，则返回空字符串。

例子：

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN

  -- 一些处理可能引起异常
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

40.7. 游标

如果不想一次执行整个命令，可以设置一个封装该命令的游标（*cursor*）， 然后每次读取几行命令结果。这么干的一个原因是在结果包含数量非常大的行时避免内存耗尽。 不过PL/pgSQL用户不必担心这个，因为 `FOR` 循环自动在内部使用一个游标以避免内存问题。 一个更有趣的用法是某个函数可以返回一个它创建的游标的引用，这样就允许调用者读取各行。 从而提供了一种从函数返回一个结果集的手段。

40.7.1. 声明游标变量

所有在PL/pgSQL里对游标的访问都是通过游标变量实现的， 它总是特殊的数据类型 `refcursor` 。 创建游标变量的一个方法是把它声明为一个类型为 `refcursor` 的变量。 另外一个方法是使用游标声明语法，像下面这样：

```
_name_ [ [ NO ] SCROLL ] CURSOR [ ( `_arguments_` ) ] FOR _query_;
```

(Oracle兼容中 `FOR` 可以用 `IS` 代替)。 如果定义了 `SCROLL`，则游标可以向后回滚；如果定义了 `NO SCROLL`， 则向后抓取的动作被拒绝；如果二者都没有定义， 那么是否进行向后取的动作会根据查询来判断。 如果有 `_arguments_`， 那么它是一个逗号分隔 `_name_``_datatype_` 列表， 这个列表定义由已给查询中的参数值来替代的name。 实际用于代换这些名字的数值将在游标打开之后声明。

例如：

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

所有这三个变量都是 `refcursor` 类型，但是第一个可以用于任何命令， 而第二个已经绑定（*bound*）了一个声明完整的命令，最后一个是绑定了一个带参数的命令。 `key` 将在游标打开的时候被代换成一个整数。 变量 `curs1` 可以称之为未绑定的， 因为它没有和任何查询相绑定。

40.7.2. 打开游标

在你使用游标检索行之前，你必需先打开它。 这是和SQL命令 `DECLARE CURSOR` 相等的操作。 PL/pgSQL有三种形式的 `OPEN` 语句， 两种用于未绑定的游标变量， 另外一种用于已绑定的游标变量。

> **Note:** 可以通过Section 40.7.4中描述的 `FOR` 语句， 在不用打开游标的情况下使用已绑定的游标。

40.7.2.1. `OPEN FOR _query_`

```
OPEN _unbound_cursorvar_ [ [ NO ] SCROLL ] FOR _query_;
```

该游标变量打开并且执行给出的查询。游标不能是已经打开的， 并且它必需是声明为一个未绑定的游标(也就是声明为一个简单的 `refcursor` 变量)。 查询必须是一条 `SELECT` 或者其它返回行的东西(比如 `EXPLAIN`)。 查询是和其它在PL/pgSQL里的SQL命令平等对待的：先代换PL/pgSQL的变量名， 而且执行计划为将来可能的复用缓存起来。 当一个PL/pgSQL变量被替换到游标查询中时， 被替换的值是在 `OPEN` 时它所具有的值。 后续的改变不会影响游标的动作， 对于一个已经绑定的游标来说， `SCROLL` 和 `NO SCROLL` 这两个选项具有相同的含义。

一个例子：

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

40.7.2.2. `OPEN FOR EXECUTE`

```
OPEN _unbound_cursorvar_ [ [ NO ] SCROLL ] FOR EXECUTE _query_string_
                                     [ USING `_expression_` [, ... ] ];
```

打开游标变量并且执行给出的查询。游标不能是已打开的， 并且必须声明为一个未绑定的游标(也就是一个简单的 `refcursor` 变量)。 命令是用和那些用于 `EXECUTE` 命令一样的方法声明的字符串表达式， 这样，就有了命令可以在两次运行间发生变化的灵活性。 参阅Section 40.10.2）这也意味着在命令字符串上不能进行变量替换。 跟 `EXECUTE` 一起，通过使用 `USING` ，参数值可以被插入到动态命令中。 对于一个已经绑定的游标来说， `SCROLL` 和 `NO SCROLL` 这两个选项具有相同的含义。

一个例子：

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
                                        || ' WHERE col1 = $1' USING keyvalue;
```

在这个例子中，表名被插入到文本查询中， 因此使用 `quote_ident()` 时要注意SQL注入。 通过 `USING` 参数对插入的 `col1` 进行比较值，因此不需要使用引号。

40.7.2.3. 打开一个绑定的游标

```
OPEN _bound_cursorvar_ [ ( [ `_argument_name_` := ] `_argument_value_` [, ...] ) ];
```

这种形式的 `OPEN` 用于打开一个游标变量， 该游标变量的命令是在声明的时候和它绑定在一起的。游标不能是已经打开的。 当且仅当该游标声明为接受参数的时候，语句中才必需出现一个实际参数值表达式的列表。 这些值将代换到命令中。

一个绑定的游标的命令计划总是认为可缓冲的，这种情况下没有等效的 `EXECUTE` 。 需要注意的是 `SCROLL` 和 `NO SCROLL` 不能在 `OPEN` 中被声明， 因为游标的滚动动作已经被定义了。

参数值可以使用*positional*或者*named*符号传递。 在位置符号中，所有的参数以顺序指定。 在命名法中，每个参数的名称使用 `:=` 声明以 从参数表达式中分开。类似于调用函数，在Section 4.3中描述， 它也允许混合位置和命名法。

例子（以上使用游标声明的例子）:

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

因为在绑定游标查询上做了变量替换，有两种方法将值传递到游标：要么 使用明确参数到 `OPEN` ， 或者隐式地在查询中引用PL/pgSQL变量。 然而，只有在绑定游标之前声明的变量将取代它。在这两种情况下 可以在 `OPEN` 时决定将被传递的值。例如， 另一种方式来获得相同的效果 如 `curs3` 上面的例子

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

40.7.3. 使用游标

一旦你已经打开了一个游标，那么你就可以用这里描述的语句操作它。

这些操作不需要发生在和打开该游标开始操作的同一个函数里。 你可以从函数里返回一个 `refcursor` 值，然后让调用者操作该游标。 在内部， `refcursor` 值只是一个包含该游标命令的活跃查询的信使的字符串名。 这个名字可以传来传去，可以赋予其它 `refcursor` 变量等等，也不用担心扰乱信使。

所有信使在事务的结尾都会隐含地关闭。 因此一个 `refcursor` 值只能在该事务结束前用于引用一个打开的游标。

40.7.3.1. `FETCH`

```
FETCH [ `_direction_` { FROM | IN } ] _cursor_ INTO _target_;
```

`FETCH` 从游标中检索下一行到目标中，目标可以是一个行变量、记录变量、逗号分隔的普通变量列表，就像 `SELECT INTO` 一样，如果下一行中没有，目标会设为NULL。如同 `SELECT INTO`， 可以使用特殊变量 `FOUND` 来检查是否检索出一个行。

`_direction_` 子句可以是任何 一个SQL FETCH命令允许的变量， 除了那些可以抓取不止一行的；形如： `NEXT`，`PRIOR`， `FIRST`， `LAST`， `ABSOLUTE` `_count_`， `RELATIVE` `_count_`， `FORWARD` 或者 `BACKWARD` 。 忽略 `_direction_` 作为声明的 `NEXT` 是相同的。`_direction_` 值需要往后移动可能会失败，除非声明的或者打开的 游标带有 `SCROLL` 选项。

`_cursor_` 必须是一个指向一个 打开的游标的 `refcursor` 变量的名字。

一个例子：

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

40.7.3.2. `MOVE`

```
MOVE [ `_direction_` { FROM | IN } ] _cursor_;
```

`MOVE` 重新定位一个游标，而不需要检索任何数据。 `MOVE` 的工作方式与 `FETCH` 及其相似，除了它只是重新定位游标并且不返回至移动到的行。 在进行 `SELECT INTO` 命令时， 声明的 `FOUND` 变量可以用来检查下一个需要移动到的行是否存在。

`_direction_` 可以是任何一个SQL FETCH 命令允许的变量，如下 `NEXT`，`PRIOR`， `FIRST`， `LAST`， `ABSOLUTE` `_count_`，`RELATIVE` `_count_`，`ALL`， `FORWARD` [`_count_` | `ALL`] 或者 `BACKWARD` [`_count_` | `ALL`]。 忽略 `_direction_` 作为声明的 `NEXT` 是相同的。`_direction_` 值需要往后移动可能会失败，除非声明的或者打开的 游标带有 `SCROLL` 选项。

例如：

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

40.7.3.3. `UPDATE/DELETE WHERE CURRENT OF`

```
UPDATE _table_ SET ... WHERE CURRENT OF _cursor_;
DELETE FROM _table_ WHERE CURRENT OF _cursor_;
```

当一个游标被定位到一个表的行上，那么通过使用该游标来识别该行， 从而进行更新或删除操作。当然，对于如何定义游标查询（特别是没有分组时）是存在一定限制的；在游标中使用 `FOR UPDATE` 是个不错的主意。更多信息可参阅DECLARE。

例如：

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

40.7.3.4. `CLOSE`

```
CLOSE _cursor_;
```

`CLOSE` 关闭支撑在一个打开的游标下面的信使。 这样就可以在事务结束之前释放资源， 或者释放掉该游标变量，用于稍后再次打开。

例如：

```
CLOSE curs1;
```

40.7.3.5. 返回游标

PL/pgSQL函数可以向调用者返回游标这个功能用于从函数里返回多行或多列， 特别是巨大的结果集。要想这么做，该函数必须打开游标并且把该游标的名字返回给调用者， 或者简单的使用指定的入口名或调用者已知的名字打开游标。 调用者然后从游标里抓取行。游标可以由调用者关闭，或者是在事务结束的时候自动关闭。

函数返回的游标名可以由调用者声明或者自动生成。 要声明一个信使的名字，只要在打开游标之前，给 `refcursor` 变量赋予一个字符串就可以了。 `refcursor` 变量的字符串值将被 `OPEN` 当作下层的信使的名字使用。 不过，如果 `refcursor` 变量是空， 那么 `OPEN` 将自动生成一个和现有信使不冲突的名字， 然后将它赋予 `refcursor` 变量。

> **Note:** 一个绑定的游标变量其名字初始化为对应的字符串值， 因此信使的名字和游标变量名同名，除非程序员在打开游标之前通过赋值覆盖了这个名字。 但是一个未绑定的游标变量初始化的时候缺省是空， 因此它会收到一个自动生成的唯一名字，除非被覆盖。

下面的例子显示了一个调用者声明游标名字的方法：

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

下面的例子使用了自动生成的游标名：

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;
-- 需要在一个事务中使用游标。
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

下面的例子显示了从一个函数里返回多个游标的方法：

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;
-- 需要在事务里使用游标。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

40.7.4. 通过游标结果进行循环

有这么一个 `FOR` 语法的变形，它允许通过游标返回的行进行迭代。如下：

```
[ <<`_label_`>> ]
FOR _recordvar_ IN _bound_cursorvar_ [ ( [ `_argument_name_` := ] `_argument_value_` [, .
    _statements_
END LOOP [ `_label_` ];
```

在声明游标变量时，它必须已经绑定到一些查询语句上，并且不能是打开状态。 `FOR` 语法会自动打开游标，并且当退出循环时自动关闭游标。只有当游标被声明要使用参数时， 必须有一列实际参数值表达式。这些值会被替换到查询中，采用如同 `OPEN` 的方式 (参阅Section 40.7.2.3)。

`_recordvar_` 变量会自动定义为 `record` 类型， 并且只存在于循环中（循环中任何的定义变量名的动作都会被忽略）。 每一个由游标返回的行都会陆续的被分配到记录变量中，然后执行循环体。

40.8. 错误和消息

利用 `RAISE` 语句报告信息以及抛出错误。

```
RAISE [ `_level_` ] '_format_' [, `_expression_` [, ... ]] [ USING `_option_` = `_express
RAISE [ `_level_` ] _condition_name_ [ USING `_option_` = `_expression_` [, ... ] ];
RAISE [ `_level_` ] SQLSTATE '_sqlstate_' [ USING `_option_` = `_expression_` [, ... ] ];
RAISE [ `_level_` ] USING _option_ = _expression_ [, ... ];
RAISE ;
```

`_level_` 选项声明了错误的严重性等级。 可能的级别有 `DEBUG` , `LOG` , `INFO` , `NOTICE` , `WARNING` , 和 `EXCEPTION` ， 默认的是 `EXCEPTION` 。 `EXCEPTION` 会抛出一个错误（强制关闭当前事务）， 而其他级别仅仅是产生不同的优先级信息。 无论是将优先级别的信息是报告给客户端，还是写到服务器日志，亦或是二者都是， 都是由log_min_messages和client_min_messages配置变量控制的。 参阅Chapter 18获取更多细节。

如果真有的话，在 `_level_` 之后， 你可以写 `_format_` ， （这必须是一个简单的字符串文本，而不是表达式）。 格式字符串声明要报告的错误信息文本。 格式字符串可以遵循插入到信息中的可选参数表达式。 在格式字符串里， `%` 被下一个可选参数的外部表现形式代替。 要表示 `%` 字符必须发出(`%%`)。

在这个例子里， `v_job_id` 的值将代替字符串中的 `%` ：

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

你可以通过在 `_option_` = `_expression_` 项后边写 `USING` 来附加额外信息到错误报告中。 每一个 `_expression_` 可以是任何字符串值表达式。 允许的 `_option_` 关键字是：

`MESSAGE`

设置错误消息文本。这个选项不能用于包含 `USING` 之前的格式字符串 的 `RAISE` 形式中。

`DETAIL`

提供一个错误详细信息。

`HINT`

提供提示信息。

`ERRCODE`

指定错误代码（SQLSTATE）用来报告，通过条件名， 如Appendix A，或直接作为 五个字符的SQLSTATE代码。

`COLUMN``CONSTRAINT` `DATATYPE` `TABLE` `SCHEMA`

提供一个相关对象名称。

该例子会强制退出事务，并返回如下提示：

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

下面两个例子在设置SQLSTATE方面具有相同的作用：

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

这是第二个 `RAISE` 语法，其中主要参数是条件名字或者要报告的SQLSTATE，比如：

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

在这个语法中， `USING` 可以来提供一个通用的错误信息，详情，或者提示。另一个较早的例子是：

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

另一个变形是 写 `RAISE USING` 或者 `RAISE``_level_` USING， 然后将其他的所有东西都放在 `USING` 列中。

最后一个 `RAISE` 变形中没有任何参数。 这种形式只能在 `BEGIN` 块的 `EXCEPTION` 字句中使用。它的作用是将正在处理的错误放到下一个封闭的块中。

> **Note:** 在PostgreSQL 9.1之前， 没有参数的 `RAISE` 被解释为 包含有活跃异常处理程序的块中重新抛出错误。 因此， `EXCEPTION` 子句嵌套在该处理器中无法抓取它，即使 `RAISE` 在嵌套的 `EXCEPTION` 子句块中。 这被认为是令人惊讶并且不兼容Oracle的PL/SQL。

如果 `RAISE EXCEPTION` 中没有声明SQLSTATE的情形名称， 那么缺省使用 `RAISE_EXCEPTION` (`P0001`)。 如果没有声明信息文本，那么缺省将情形名称或SQLSTATE作为信息文本。

> **Note:** 当通过SQLSTATE编码声明一个错误代码时，你不能限制预定义错误代码， 但是可以选择任何由五个数字和/或者大写ASCII字母组成的错误代码， 而不是 `00000` 。建议避免抛出以三个零结尾的错误代码，因为 这些是类别码并且只能通过捕获整个类别来获取。

40.9. 触发器过程

40.9.1. 对数据变化的触发

PL/pgSQL可以用于定义触发器过程。 一个触发器过程是用 `CREATE FUNCTION` 命令创建的， 创建的形式是一个不接受参数并且返回 `trigger` 类型的函数。 请注意该函数即使在 `CREATE TRIGGER` 声明里声明为准备接受参数， 它也必需声明为无参数，因为触发器的参数是通过 `TG_ARGV` 传递的(下面有描述)。

在一个PL/pgSQL函数当做触发器调用的时候， 系统会在顶层的声明段里自动创建几个特殊变量。有如下这些：

`NEW`

数据类型是 `RECORD` ； 该变量为行级触发器中的 `INSERT` / `UPDATE` 操作存储新数据行。 在语句级别的触发器里这个变量以及 `DELETE` 操作未赋值。

`OLD`

数据类型是 `RECORD` ；该变量为行级触发器中的 `UPDATE` / `DELETE` 操作存储旧数据行。 在语句级别的触发器里以及对 `INSERT` 动作，这个变量未赋值。

`TG_NAME`

数据类型是 `name` ；该变量包含实际触发的触发器名。

`TG_WHEN`

数据类型是 `text` ；是一个由触发器定义决定的字符串 (`BEFORE` ， `AFTER` 或者 `INSTEAD OF`)。

`TG_LEVEL`

数据类型是 `text` ；是一个由触发器定义决定的字符串(`ROW` 或者 `STATEMENT`)。

`TG_OP`

数据类型是 `text` ；是一个说明激活触发器的操作的字符串 (`INSERT` ， `UPDATE` ， `DELETE` 或者 `TRUNCATE`)。

`TG_RELID`

数据类型是 `oid` ；是激活触发器调用的表的对象标识(OID)。

`TG_RELNAME`

数据类型是 `name` ；是激活触发器调用的表的名称。 反对使用，并会在将来的版本中消失，推荐使用 `TG_TABLE_NAME` 。

`TG_TABLE_NAME`

数据类型是 `name` ；是激活触发器调用的表的名称。

`TG_TABLE_SCHEMA`

数据类型是 `name` ；是激活触发器调用的表的模式名。

`TG_NARGS`

数据类型是 `integer` ；是在 `CREATE TRIGGER` 语句里面赋予触发器过程的参数的个数。

`TG_ARGV[]`

数据类型是 `text` 的数组；是 `CREATE TRIGGER` 语句里的参数。下标从0开始记数。 非法下标(小于0或者大于等于 `tg_nargs`)导致返回一个NULL值。

一个触发器函数必须返回 `NULL` 或者是 一个与激活触发器运行的表的记录/行结构完全相同的数据。

因 `BEFORE` 触发的行级别触发器可以返回一个NULL， 告诉触发器管理器忽略对该行剩下的操作，也就是说，随后的触发器将不再执行， 并且不会对该行产生 `INSERT` / `UPDATE` / `DELETE` 动作)。 如果返回了一个非NULL的行，那么将继续对该行数值进行处理。请注意， 返回一个和原来的 `NEW` 不同的行数值将修改那个将插入或更新的行 因此，如果想在没有修改行值的同时成功的执行触发器动作，那么需要返回 `NEW` （或等价的）。 为了修改行存储，可以用一个值直接代替 `NEW` 里的某个数值并且返回之， 或者也可以构建一个全新的记录/行再返回。 在 `DELETE` 上的before触发器的情况下， 返回值没有直接的影响，但是它不得不是非null以允许触发器操作继续执行。 请注意 `DELETE` 触发器中 `NEW` 是null，因此 返回往往是不明智的。 `DELETE` 触发器通常情况返回 `OLD` 。

`INSTEAD OF` 触发器（总是行级触发器，并且只能用于视图）可以返回null标记他们 不执行任何更新，并且应该忽略这些行操作的剩余部分（比如，随后的触发器不会被触发，并且 为了周围的 `INSERT` / `UPDATE` / `DELETE` 在受影响的行状态下不计算行）。 另外应该返回一个空值，用来标记触发器执行所需要的操作。为了 `INSERT` 和 `UPDATE` 操作，返回值应是 `NEW` ， 这个触发器函数可以修改以支持 `INSERT RETURNING` 和 `UPDATE RETURNING` （这也将影响传递到任何随后触发器的行值）。 为了 `DELETE` 操作，返回值应是 `OLD` 。

一个 `AFTER` 行级别的触发器或者 `BEFORE` 或者 `AFTER` 语句级别的触发器 返回值将总是被忽略； 它们也可以返回NULL来忽略返回值。不过， 任何这种类型的触发器仍然可以通过抛出一个错误来退出整个触发器操作。

Example 40-3显示了一个PL/pgSQL 写的触发器过程的例子。

Example 40-3. PL/pgSQL触发器过程

下面的示例触发器的作用是：任何时候表中插入或更新了行，当前的用户名和时间都记录入行中。 并且它保证给出了雇员名称并且薪水是一个正数。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN

-- 检查是否给出了empname和salary
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

-- 必须付账给谁?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

-- 记住何时何人的薪水被修改了

        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

另外一个向表里记录变化的方法涉及创建一个新表，然后为后来发生的每次插入、更新或者删除动作保存一行。这个方法可以当作对一个表的审计。Example 40-4显示了一个PL/pgSQL写的审计触发器过程的例子。

Example 40-4. PL/pgSQL审计触发器过程

这个例子触发器保证了在 `emp` 表上的任何插入、更新、删除动作都被记录到了 `emp_audit` 表里(也就是审计)。当前时间和用户名会被记录到数据行里，以及还有执行的操作。

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN

--
        -- 在emp_audit里创建一行，反映对emp的操作，
        -- 使用特殊变量TG_OP获取操作类型。
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

先前例子的一个变化使用连接主表到审计表的视图， 显示上次修改的每个项。这个方法还记录了改变表的完整审计追踪，但是 也提出了审计追踪的简单视图，显示来源于每项审计追踪的最后修改的时间戳。 Example 40-5显示了PL/pgSQL 中视图上的审计触发器的例子。

Example 40-5. 审计PL/pgSQL视图触发器程序

这个例子使用视图上的一个触发器更新，并且 确保任何插入，更新或删除视图中的一行被记录（即，审核）在 `emp_audit` 表中。 当前时间和用户名被记录，连同执行操作类型，而且视图显示每一行的最后修改时间。

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN

--
        -- 在emp上执行所需操作，并且在emp_audit中创建一行以反映emp所做的变化。
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

触发器的一个用途是维持另外一个表的概要。 生成的概要可以用于在某些查询中代替原始表(通常可以大大缩小运行时间)。 这个技巧经常用于数据仓库，这个时候， 需要测量的表(叫事实表)可能会非常巨大。 Example 40-6演示了一个 PL/pgSQL触发器过程的例子， 它为某个数据仓库的一个事实表维护一个概要表。

Example 40-6. 一个维护概要表的PL/pgSQL触发器过程

下面的模式有一部分 是基于数据仓库工具里面的*Grocery Store*例子。

```
--
-- 主表 - 时间维以及销售事实。
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);
--
-- 摘要表-根据时间的销售。
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);
--
-- 在UPDATE，INSERT，DELETE的时候更新概要字段的函数和触发器
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

 -- 计算增/减量
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

-- 禁止改变 time_key 的更新
            -- (可能并不是很强制，因为 DELETE + INSERT 是大多数可能产生的修改)。

            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                      OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
```

```
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;

 --用新数值插入或更新概要行。

        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- do nothing
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

40.9.2. 事件触发器

PL/pgSQL用于定义事件触发器。PostgreSQL 要求作为事件触发器调用的程序必须声明为无参函数，并且返回 `event_trigger` 类型。

当PL/pgSQL函数作为事件触发器调用时， 在顶层自动创建一些特殊变量，他们是：

`TG_EVENT`

数据类型 `text`;表示事件的字符串触发触发器。

`TG_TAG`

数据类型 `text`;包含命令标签的变量触发的触发器。

Example 40-7显示PL/pgSQL中的 事件触发器程序例子。

Example 40-7. PL/pgSQL事件触发器程序

这个例子触发器每次执行可支持命令时简单触发 `NOTICE` 消息。

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

40.10. 在后台下的PL/pgSQL

本节讨论PL/pgSQL用户知道的比较重要的一些实现细节。

40.10.1. 变量替换

在PL/pgSQL函数内的SQL语句和表达式 可以参考变量和函数的参数。在后台，PL/pgSQL替代这些参考查询参数。 参数只会按照句法允许一个参数或列引用的地方被取代。作为一个极端的例子， 考虑不好的编程风格的这个例子：

```
INSERT INTO foo (foo) VALUES (foo);
```

`foo` 第一次出现一定在语法上是表名字，所以它不会被取代，即使函数有一个可变名 `foo` 。第二次发生必须是表列名称，所以它也不会被取代。 只有第三次发生是参考函数变量的一个候选。

> **Note:** 9.0之前的PostgreSQL版本可能尝试所有三种情况中替换变量，导致语法错误。

由于变量的名字语法上和表列名字没有什么不同，参考表的语句中有模糊：它是一个给定的名字意味着引用一个表列，或一个变量？ 让我们改变以往的例子

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

在这里， `dest` 和 `src` 必须是表名，并且 `col` 必须是 `dest` 的列，但是 `foo` 和 `bar` 可能是函数变量或者 `src` 的列。

默认情况下，如果在一个SQL语句中的名字 可以参考一个变量或表列，则PL/pgSQL将报告错误。 你可以通过重命名变量或列，或限定不明确的引用，或者告诉PL/pgSQL 说明更喜欢哪个来解决这样的问题。

最简单的解决方案是重命名变量或列。一个常见的编码规则是使用 PL/pgSQL变量的不同命名惯例而不是你使用列名称。 比如，如果你一贯地命名函数变量 `v_``_something_` ， 然而你的列没有以 `v_` 开头命名，不会发生冲突。

另外你可以限定含糊的引用以使得它们明确。 在上面的例子中， `src.foo` 将是表列的明确参考。 为了创建明确的引用变量， 在标记块声明它并且使用块标签（参阅Section 40.2）。比如，

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

这里 `block.foo` 意味着变量， 即使在 `src` 中有一列 `foo` 。 函数的参数，以及特殊变量如 `FOUND` ， 可以满足 函数的名字，因为他们在使用函数名标记的外部块中隐式声明。

有时修复在PL/pgSQL编码主体下所有不明确的引用是不切实际的。 在这种情况下，你可以指定PL/pgSQL应该解决不明确的引用，作为变量 （即兼容PostgreSQL 9.0之前的PL/pgSQL的行为）， 或作为表列（与其他一些系统兼容，如Oracle）

要改变在系统范围基础上的这种行为，设置配置参数 `plpgsql.variable_conflict` 为 `error` , `use_variable` 或者 `use_column` 之一（ `error` 是出厂缺省值）。 此参数会影响PL/pgSQL函数中语句后续的编译，但不是在当前会话中已编译的语句。 由于更改此设置可以导致PL/pgSQL函数行为意想不到的变化，它只能由超级用户改变。

你也可以在功能分析的基础上设定行为，通过在函数文本的开始处插入这些特殊的命令之一：

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

这些命令只影响写入的函数， 并且重写 `plpgsql.variable_conflict` 的设置。例如：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

在 `UPDATE` 命令中， `curtime` , `comment` , 和 `id` 将引用函数的变量和 `users` 是否具有这些名称列的参数。 请注意我们必须限定到 `WHERE` 子句 `users.id` 的引用以使得它引用表列。 我们没有必要限定引用到 `comment` 作为 `UPDATE` 列表中的目标， 因为语法上必定是 `users` 的列。 我们可以写不依赖于这种方式的 `variable_conflict` 设置的同样函数：

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

变量代换不会发生在给定 `EXECUTE` 或者它的变种之一的命令字符串中。 如果你需要插入一个不同的值到这个命令中， 执行它作为构建字符串值的一部分， 或使用 `USING`， 正如 Section 40.5.4说明的。

变量替换目前只能在 `SELECT`, `INSERT`, `UPDATE` 和 `DELETE` 命令中运行，因为主要的SQL引擎允许这些命令中的查询参数。 为了使用其他语句类型中非恒定的名称或值 （一般称为实用语句）， 你必须构建实用语句作为字符串并且 `EXECUTE` 它。

40.10.2. 计划缓存

PL/pgSQL解释器解析函数的源 文本并且第一次函数被调用时（每个会话中）产生一个内部二进制指令树。 指令树充分翻译PL/pgSQL语句结构， 但个别SQL表达式和在函数中使用的SQL命令不是立即翻译。

在函数中首先执行每个表达式和SQL命令，PL/pgSQL 解释器解析并且分析命令以创建预备语句，使用SPI管理的 `SPI_prepare` 函数。 随后访问表达式或命令重新使用事先准备好的语句。 因此，带有条件编码路径的函数很少被访问将不会产生分析不在当前会话中执行的命令的开销， 一个缺点是在一个特定的表达或命令中的错误 不能被检测到直到执行达到函数部分（琐碎的语法错误在初步解析传递期间将被检测到，但是任何更深的东西将不会被检测到直到执行为止。）

PL/pgSQL（或者更准确的说，SPI管理者）可以 尝试与任何特别已准备语句相关的缓存执行计划。 如果没有使用缓存计划， 那么在每次访问语句中产生一个新的执行计划，并且 当前的参数值（即，PL/pgSQL 变量值）可以用来优化选择方案。 如果语句没有参数，或是执行多次，SPI管理者 将考虑创建*generic*计划不依赖于特定的参数值，并且缓存再利用。 只有执行计划对PL/pgSQL变量中引用的值不太敏感时，往往会发生。 如果是，每次生成一个计划都是净赢。 参阅PREPARE获得更多有关预备语句行为信息。

因为PL/pgSQL保存已预备好语句并且有时以这种方式执行计划， 直接出现在PL/pgSQL函数中的SQL命令必须查阅相同表和每个执行列； 也就是说， 你不能使用参数作为SQL命令的表或列的名字。 为了应对这个限制， 你可以使用PL/pgSQL `EXECUTE` 语句构建动态命令；以执行新的解析分析和每个执行上构建新的执行计划为代价。

记录变量的可变性质提出连接中的另一个问题。 当在表达式或语句中使用记录变量字段时，该字段数据类型必须不能从函数的一个调用到下一个改变， 因为当表达式第一个到达时使用目前数据类型分析每个表达式。 `EXECUTE` 必要时可以用于解决这个问题。

如果相同函数作为多个表的触发器使用， PL/pgSQL为了每个表独立地准备并且缓存声明；即，有一个触发器函数和表组合的高速缓存，而不只是为每个函数。 这解决了一些数据类型不同的问题；例如，一个触发器函数可以使用命名 `key` 的列 成功运行，即使发生不同的表中有不同的类型。

同样，具有多态性参数类型的函数 有一个他们被调用的实际参数类型的每个组合的单独声明缓存， 所以该数据类型差异不会导致意外失败。

语句缓存有时会对时间敏感值的解释有令人惊讶的影响。 例如在这两个函数要做的内容之间有区别：

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

和

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

在 `logfunc1` 的情况下，该 PostgreSQL主解析器知道当 分析 `INSERT` 时 字符串 `'now'` 应解释为 `时间戳`， 因为 `logtable` 目标列是那种类型。 因此，当分析 `INSERT` 时， `'now'` 将被转换为 `timestamp` 常量， 然后在会话的整个生命周期中用于 `logfunc1` 的所有调用。 不用说，这不是程序员希望的。 一个更好的办法是使用 `now()` 或者 `current_timestamp` 函数。

在 `logfunc2` 的情况下， PostgreSQL主解析器并不知道 `'now'` 应该成为什么类型，因此它返回包含字符串 `now` 类型 `text` 的数据值。 随后分配给局部变量 `curtime` 期间， PL/pgSQL解析器通过调用 `text_out` 和 `timestamp_in` 转换函数将这个字符串转换为 `timestamp` 类型， 因此，作为编程期望每次执行时更新计算时间戳。 尽管这正如预期的那样发生，这不是非常有效的， 所以 `now()` 函数的使用仍然会是一个更好的主意。

40.11. 开发PL/pgSQL的一些提示

用PL/pgSQL做开发的一个好方法是简单地使用文本编辑器创建函数， 然后在另外一个控制台里，用psql加载这些函数。 如果你用这种方法，那么用 `CREATE OR REPLACE FUNCTION` 写函数是个好主意。 这样，重读文件就可以更新函数定义。比如：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

在运行psql的时候， 可以用下面命令加载或者重载函数定义文件:

```
\i filename.sql
```

然后马上发出SQL命令测试该函数。

另外一个开发PL/pgSQL程序的好方法是使用一种支持 过程语言开发的GUI工具。 比如pgAdmin，当然还有其它的。这些工具通常提供了一些很有用的功能， 比如逃逸单引号使得重建和调试函数更简单等。

40.11.1. 引号标记处理

PL/pgSQL函数的代码都是在 `CREATE FUNCTION` 里以一个字符串文本的方式声明的。 如果你用两边包围单引号的常规方式写字符串文本，那么任何函数体内的单引号都必须写双份； 类似的是反斜杠也必须双份。双份引号非常乏味，在更复杂的场合下，代码可能会让人难以理解， 因为你很容易发现自己需要半打甚至更多相连的引号。 建议你用"dollar-quoted"的字符串文本来写函数体。 (参阅Section 4.1.2.4)。使用美元符界定的时候， 你从不需要对任何引号写双份， 只需要为每层引号包围嵌套选择一个不同的美元符号包围分隔符即可。 比如，你可能这么写 `CREATE FUNCTION` 命令：

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

在这个函数体中，可以在SQL命令里使用单引号包围文本字符串， 用 `$$` 分隔那些SQL命令的片断。 如果你需要对包含 `$$` 的文本进行引号包围，可以使用 `$Q$` 等等。

下表展示了不使用美元符界定的时候该如何写单引号。 把美元符引号之前的引号包围的代码转换成某种可以理解的形式时， 应该会用得上。

1个单引号

开始/结束函数体，比如：

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

在函数体内部的任何位置，问号都必须成对出现。

2个单引号

对于函数体内的字符串文本，比如：

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

在美元符界定的方法里，你只要写:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

两种情况都是PL/pgSQL分析器期望看到的东西。

4个单引号

如果你在函数体中的字符串里面需要一个单引号，比如：

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

`a_output` 的值将是 `AND name LIKE 'foobar' AND xyz` 。

使用美元符界定的方法应该这样写:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

注意，这样的美元符界定的分隔符并不是只有 `$$` 。

6个单引号

如果一个在函数体中的字符串内的单引号与该字符串常量结尾前后相连，比如：

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

`a_output` 的值将是 `AND name LIKE 'foobar'` 。

用美元符界定的方法则为是:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10个单引号

如果你想要在字符串常量里有两个单引号(它们在一起是8个了)， 并且这两个单引号和该字符串常量的结尾相连(又加2个)。 可能只有在写一个生成其它函数的函数的时候， 像Example 40-9里那样。比如：

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

`a_output` 的值将是:

```
if v_... like ''...'' then return ''...''; end if;
```

使用美元符界定的方法应该这样写:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

假设我们只需要在 `a_output` 里放单引号，因为在使用前它会被重新引号包围。

40.12. 从Oracle PL/SQL进行移植

本节解释了PostgreSQL的PL/pgSQL 和Oracle的PL/SQL语言之间的差别， 希望能对那些从Oracle®向 PostgreSQL移植应用的人有所帮助。

PL/pgSQL与PL/SQL在许多方面都非常类似。 它是一种块结构的，祈使语气(命令性)的语言并且必须声明所有变量。 赋值、循环、条件等都很类似。在从PL/SQL向 PL/pgSQL移植的时候必须记住一些事情：

- 如果一个SQL命令中使用的名字是一个表中的列名，或者是一个函数中变量的引用， 那么PL/SQL会将它当作一个变量名。 这对应的是PL/pgSQL的 `plpgsql.variable_conflict` = `use_column` 动作（不是默认动作）， 参考Section 40.10.1中的描述。 首先，最好是避免这种模糊的方式，但如果不得不移植一个依赖于该动作的大量的代码， 那么设置 `variable_conflict` 是个不错的主意。
- 在PostgreSQL里，函数体必须写成字符串文本， 因此你需要使用美元符界定或者逃逸函数体里面的单引号(见Section 40.11.1)。
- 应该用模式把函数组织成不同的组，而不是用包。
- 因为没有包，所以也没有包级别的变量。这一点有时候挺讨厌。 你可以在临时表里保存会话级别的状态。
- 带有 `REVERSE` 的整数的 `FOR` 循环的工作模式是不一样的： PL/SQL中是从第二个数向第一个数倒计， 而PL/pgSQL是从第一个数想第二个数倒计， 因此在移植时，需要交换循环边界。 不幸的是这种不兼容性是不太可能改变的（参阅Section 40.6.3.5）。
- 遍历查询的 `FOR` 循环（而不是循环游标）同样有不同的工作模式： 必须已经声明了目标变量，在这一点上PL/SQL通常是隐式的声明。 这样做的优点是，在退出循环后，仍然可以获得变量值。
- 在使用游标变量方面，存在一些记数法差异。

40.12.1. 移植样例

Example 40-8演示了 如何从PL/SQL向PL/pgSQL移植一个简单的函数。

Example 40-8. 从PL/SQL向PL/pgSQL移植一个简单的函数

下面是一个Oracle PL/SQL函数:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

让我们读一遍这个函数然后看PL/pgSQL与之的不同：

- 在函数原型里的 `RETURN` (不是函数体里的)关键字到了PostgreSQL里就 是 `RETURNS` 。还有， `IS` 变成 `AS` ， 并且你还需要增加一个 `LANGUAGE` 子句， 因为PL/pgSQL并非唯一可用的函数语言。
- 在PostgreSQL里，函数体被认为是一个字符串文本， 所以你需要使用单引号或者美元符界定它， 这个包围符代替了Oracle 最后的那个 `/` 。
- 在PostgreSQL里没有 `show errors` 命令， 不需要这个命令是因为错误是自动报告的。

下面是这个函数移植到PostgreSQL之后的样子:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Example 40-9演示了如何移植一个创建另外一个函数的函数的方法， 以及演示了如何处理引号逃逸的问题。

Example 40-9. 从PL/SQL向PL/pgSQL移植一个创建其它函数的函数

下面的过程从一个 `SELECT` 语句中抓取若干行， 然后为了提高效率，又用 `IF` 语句中的结果制作了一个巨大的函数。

这是Oracle版本:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
                 v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

下面是这个函数在PostgreSQL里面的样子：

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

请注意函数体是如何独立制作并且传递给 `quote_literal`，对其中的单引号复制双份的。 需要这个技巧是因为无法使用美元符界定定义新函数： 没法保证 `referrer_key.key_string` 字段过来的字符串会解析成什么样子。 可以假设 `referrer_key.kind` 是只有 `host`，`domain` 或者

`url`，但是 `referrer_key.key_string` 可能是任何东西，特别是它可能包含美元符。 这个函数实际上是对原来 Oracle 版本的一个改进， 因为如果在 `referrer_key.key_string` 或者 `referrer_key.referrer_type` 包含单引号的时候， 它不会生成有毛病的代码。

Example 40-10演示了如何移植一个带有 `OUT` 参数和字符串处理的函数。 PostgreSQL里面没有内置 `instr` 函数，但是你可以用其它函数的组合来绕开它。 在Section 40.12.3里有一个 PL/pgSQL 的 `instr` 实现，你可以用它让你的移植变得更简单些。

Example 40-10. 从PL/SQL向 PL/pgSQL移植一个字符串操作和 `OUT` 参数的过程

下面的Oracle PL/SQL过程用于分析一个URL并且返回若干个元素(主机、路径、命令)。

下面是Oracle版本:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
v_host OUT VARCHAR,  -- 这个变量是要传回的
    v_path OUT VARCHAR,  -- 这个也是
    v_query OUT VARCHAR) -- 还有这个
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

下面就是把这个过程翻译成PL/pgSQL可能的样子：

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
v_host OUT VARCHAR,  -- 这个将被传回
    v_path OUT VARCHAR,  -- 这个也传回
    v_query OUT VARCHAR) -- 还有这个
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

这个函数可以这么用:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 40-11演示了如何一个使用各种Oracle专有特性的过程。

Example 40-11. 从PL/SQL向PL/pgSQL移植一个过程

Oracle版本：

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;<a name="CO.PLPGSQL-PORTING-PRAGMA">**(1)**</a>
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;<a name="CO.PLPGSQL-PORTING-LOCKTABLE">**(2)**</

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock<a name="CO.PLPGSQL-PORTING-COMMIT">**(3)**</a>
        raise_application_error(-20000,
                 'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
    END;
    COMMIT;
END;
/
show errors
```

像这样的过程可以很容易用返回 `void` 的函数移植到PostgreSQL里。 对这个过程特别感兴趣是因为它可以教一些东西：

(1)

在PostgreSQL里没有 `PRAGMA` 语句。

(2)

如果你在PL/pgSQL里做一个 `LOCK TABLE` ， 那么这个锁在调用该命令的事务完成之前将不会释放。

(3)

你不能在PL/pgSQL函数里发出 `COMMIT` 。 函数是在外层的事务里运行的，因此 `COMMIT` 蕴涵着结束函数的执行。 不过，在这个特殊场合下，这是不必要的了， 因为 `LOCK TABLE` 获取的锁将在抛出错误的时候释放。

下面是把这个过程移植到PL/pgSQL里的一种方法：

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running';<a name=
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN <a name="CO.PLPGSQL-PORTING-EXCEPTION">**(2)**</a>
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

(1)

`RAISE` 的语法和Oracle的类似语句差别相当明显。 尽管 `RAISE``_exception_name_` 运行的基本情况相似。

(2)

PL/pgSQL里支持的异常的名字和Oracle的不同。 内置的异常名要大的多(参阅Appendix A)。目前还不能声明用户定义的异常名。

整个过程和Oracle的等效的主要的功能型差别是， 在 `cs_jobs` 上持有的排他锁将保持到调用的事务结束。 同样，如果调用者后来退出(比如说因为错误)，这个过程的效果将被回滚掉。

40.12.2. 其它注意事项

本节解释几个从Oracle PL/SQL函数向PostgreSQL 移植的几个其它方面的事情。

40.12.2.1. 异常后的隐含回滚

在PL/pgSQL里，如果一个异常被 `EXCEPTION` 子句捕获， 那么所有自这个块的 `BEGIN` 以来的数据库改变都会被自动回滚。 也就是说，这个行为等于你在Oracle里的：

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

如果你在翻译使用 `SAVEPOINT` 和 `ROLLBACK TO` 的Oracle过程， 那么你的活儿很好干：只要省略 `SAVEPOINT` 和 `ROLLBACK TO` 即可。 如果你要翻译的过程使用了不同的 `SAVEPOINT` 和 `ROLLBACK TO` ，那么就需要想想了。

40.12.2.2. `EXECUTE`

PL/pgSQL版本的 `EXECUTE` 类似PL/SQL运转， 不过你必须记住要像Section 40.5.4里描述的那样 用 `quote_literal` 和 `quote_ident` 。 如果你不用这些函数，那么像 `EXECUTE 'SELECT * FROM $1';` 这样的构造是不会运转的。

40.12.2.3. 优化PL/pgSQL函数

PostgreSQL给你两个创建函数的修饰词用来优化执行："volatility" (易变的，在给出的参数相同时， 函数总是返回相同结果)和"strictness" (严格的，如果任何参数是NULL，那么函数返回NULL)。 参考CREATE FUNCTION的手册获取细节。

如果要使用这些优化属性，那么你的 `CREATE FUNCTION` 语句可能看起来像这样：

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

40.12.3. 附录

本节包含Oracle兼容的 `instr` 函数，你可以用它简化你的移植过程。

```
--
-- 模拟 Oracle 概念的 instr 函数
-- 语法: instr(string1, string2, [n], [m])  这里的 [] 表示可选参数
--
-- 从 string1 的第 n 个字符开始寻找 string2 的第 m 个出现。
-- 如果 n 是负数, 则从后向前着。如果没有传递 m , 假定为 1(从第一个字符开始找)。
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
```

```
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;
```

```
        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chapter 41. PL/Tcl - Tcl 过程语言

Table of Contents

PL/Tcl 是一种用于PostgreSQL数据库系统的可加载的过程化语言， 它让我们可以用Tcl 语言来书写函数和触发器过程。

41.1. 概述

PL/Tcl 提供 C 语言里面函数开发者所拥有的大多数功能，只有一点点限制除外， 另外Tcl还可以使用强大的字符串处理库。

好的限制是，所有东西都是在一个安全的 Tcl 解释器里面运行的。 除了一个有限的 Tcl 安全命令集外，只有很少的几个命令可以通过 SPI 访问数据库以及通过 `elog()` 生成错误信息。不像 C 函数那样， Tcl 没有办法访问数据库后端内部或者获得 OS 级的PostgreSQL 服务器进程的权限。因此，任何非特权的数据库用户都可以被允许使用这种语言。

另外的实现级限制是 Tcl 过程不能用于创建新数据库类型的输入/输出函数。

有时候需要写一些不受安全 Tcl 限制的 Tcl 函数，比如，可能需要一个可以发送邮件的 Tcl 函数。 要处理这样的问题，有一个PL/Tcl的变种，叫 `PL/TclU` (不可信的 Tcl)。 这个语言和PL/Tcl 是完全一样的，只不过使用了一个完整的 Tcl 解释器。如果你使用了 *PL/TclU*，那么你必须把它安装成一种不可信的过程语言， 这样只有数据库超级用户可以用它创建函数。 PL/TclU函数的作者必须注意： 你写的函数一定不要做任何预算外的事情，因为它能干所有数据库管理员能干的事情。

如果在安装过程的配置步骤中声明了 Tcl 支持，那么PL/Tcl和PL/TclU 的调用处理器是在制作时自动制作并安装到PostgreSQL库目录中去的。 要在某个特定的数据库中安装PL/Tcl和/或PL/TclU， 那么你可以使用 `CREATE EXTENSION` 命令或 `createlang` 程序。 比如 `createlang pltcl` `_dbname_` 或 `createlang pltclu` `_dbname_` 。

41.2. PL/Tcl 函数和参数

要用PL/Tcl语言创建一个函数，使用标准的CREATE FUNCTION语法：

```
CREATE FUNCTION _funcname_ (_argument-types_) RETURNS _return-type_ AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU是一样的，除了语言应该声明为 `pltclu` 之外。

函数体就是一段 Tcl 代码。当在一个查询里面调用这个函数，参数是作为变量 `$1` ... `$``_n_` 传递给 Tcl 脚本的。 结果是用通常的方法从 Tcl 代码中返回的，就是用一个 `return` 语句。

比如，一个简单的返回两个整数值中较大值的函数可以这样定义：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

请注意 `STRICT` 子句，它让我们可以不用考虑输入为 NULL 的情况： 如果传递了一个 NULL ，该函数实际上就不会被调用，而只是自动返回一个 NULL 结果。

如果是一个不严格的函数且一个参数的实际值是 NULL ，那么对应的 `$``_n_` 变量将被设置为一个空字符串。要检测一个特定的参数是否为 NULL ，可以使用 `argisnull` 函数。 比如，假设要求 `tcl_max` 在一个参数为 null 而另外一个为非 null 时返回非 null 参数， 而不是 null ：

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

如上所述，要从 PL/Tcl 函数中返回一个 NULL 数值，可以执行 `return_null` 。 不管函数是否严格，都可以这么做。

复合类型的参数是当做 Tcl 数组传递给函数的。数组中的元素名字就是复合类型里的属性名字。 如果在传递的行中某个属性为 NULL 数值，那它就不会在数组中出现。下面是一个例子：

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

目前没有返回复合类型结果的支持。也不支持返回结果集。

PL/Tcl目前还不是完全支持域类型：它看待域类型和下层的标量类型是一样的。 这就意味着与域关联的约束将不会被强制。对于函数参数，这不是什么问题， 但是如果你把PL/Tcl函数声明为返回一个域类型，那么就有危险。

41.3. PL/Tcl里的数据值

提供给 PL/Tcl 函数脚本的参数值都只是转换成文本形式的输入参数(就像它们用 `SELECT` 语句显示出来的那样)。相反， `return` 可以用任何可以为函数所声明的返回类型接受的输入格式的字符串。因此，在 PL/Tcl 函数里， 所有数值只是文本字符串。

41.4. PL/Tcl里的全局量

有时候在两次过程函数调用或者不同的函数之间保存一些全局数据是非常有用的。 在PL/Tcl里实现这个目标相当容易，但是这里有一些限制必须熟悉。

由于安全原因，PL/Tcl通过一个在单独的Tcl解释器里的SQL角色执行函数调用。 通过有另外一个用户的PL/Tcl函数行为的用户阻止了偶然的或恶意的干扰。 每个这样的干扰对于任意的"全局" Tcl变量将有它自己的值。 因此，当且仅当他们通过同一个SQL角色执行时，两个PL/Tcl函数才分享相同的全局变量。 在一个多个SQL角色下的单个会话执行脚本的应用中（通过`SECURITY DEFINER`函数， 使用`SET ROLE`等等）可能需要采取明确的步骤确保PL/Tcl函数可以分享数据。 要做到这点，确保可以通讯的函数属于同一个用户，并且标识它们`SECURITY DEFINER`。 当然必须注意这种函数不能用来做任何计划外的动作。

所有在一个会话中使用的PL/TclU函数在同一个Tcl解释器里执行，不同于用于PL/Tcl函数的解释器。 所以全局数据自动在PL/TclU函数间分享。没有考虑安全风险，因为所有PL/TclU函数在同一个可靠地级别执行， 即数据库超级用户。

为了保护 PL/Tcl 过程相互之间不至于互相干扰，每个过程可以通过`upvar`命令访问一个全局数组。此变量的全局名称是过程的内部名称，其局部名称是`GD`。 建议使用`GD`作为函数的永久私有状态数据的存储。 而把普通的 Tcl 全局变量只用于那些你想在多个过程之间共享的变量。 （请注意，`GD`数组只在一个特定的解释器里是全局的，所以它们没有绕开上面提到的安全限制。）

一个使用`GD`的例子在下面的`spi_execp`例子里显示。

41.5. 在PL/Tcl里访问数据库

在 PL/Tcl 过程体里有下面的命令可以用于访问数据库：

`spi_exec` ` ?-count` *n* `?` `?-array` *name* `?` *command* `?` *loop-body*`?

执行一个以字符串形式给出的 SQL 查询。查询中的错误会导致抛出一个错误。否则，`spi_exec` 的返回值是命令处理的行数(选出、插入、更新、删除)， 如果该命令是一个功能性语句则返回零。另外，如果查询是一条 `SELECT` 语句， 那么选出的字段值按照下面描述的方法放在 Tcl 变量中。

可选的 `-count` 值告诉 `spi_exec` 在该查询中处理的最大的行数。 其效果和把查询设置为一个游标，然后说 `FETCH` `_n_` 是一样的。

如果查询是一个 `SELECT` 语句，那么其结果列的数值将放在用各字段名命名的 Tcl 变量中。 如果给出了 `-array` 选项，那么字段值将放到这个命名的相关数组中，字段名用做数组索引。

如果查询是 `SELECT` 语句并且没有给出 `_loop-body_` 脚本， 那么只有结果的头一行会存储到 Tcl 变量中；如果还有其它行的话，将会被忽略。 如果查询没有返回任何行，那么不会存储什么数据(这个情况可以通过检查 `spi_exec` 的结果来判断)。比如：

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

将设置 Tcl 变量 `$cnt` 设为系统表 `pg_proc` 中的行数。

如果给出了可选的 `_loop-body_` 参数，那么它就是一小段 Tcl 脚本， 它会为查询结果中的每一行执行一次(注意：如果给出的查询不是 `SELECT` ， 那么忽略 `_loop-body_`)。在每次迭代之前， 当前行的字段的数值都存储到 Tcl 变量中去了。比如：

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

将为 `pg_class` 的每一行打印一行日志信息。这个特性和其它 Tcl 循环构造的运做方式类似；特别是 `continue` 和 `break` 在循环体中的作用和平常是一样的。

如果一个查询结果的某个字段是 NULL ， 那么其目标变量就是"unset"而不会设置上什么东西。

`spi_prepare` `_query_` `_typelist_`

为后面的执行准备并保存一个查询规划。保存的规划的生命期就是当前会话的生命期。

查询可以使用参数，这些参数是规划实际执行的时候提供的数值的占位符。在查询字符串里， 用符号 `$1` ... `` $``_n_ `` 引用各个参数。 如果查询使用了参数，那么参数类型名必需以一个 Tcl 列表的形式给出。如果没有使用参数，那么给 `_typelist_` 写一个空列表。

`spi_prepare` 的返回值是一个可以在随后的 `spi_execp` 调用中使用的查询 ID 。参阅 `spi_execp` 获取例子。

`` spi_execp` ?-count `` *n* `? ?-array` *name* `? ?-nulls` *string* `?` *queryid* `?` *value-list* `? ?` *loop-body*`?

执行一个前面用 `spi_prepare` 准备的查询。 `_queryid_` 是 `spi_prepare` 返回的 ID 。如果该查询引用了参数，那么必需提供一个 `_value-list_` ： 这是一个 Tcl 列表，里面包含那些参数的实际数值。这个列表的长度必需和前面给 `spi_prepare` 提供的参数类型列表的长度一样长。 如果查询没有参数，那么省略 `_value-list_` 。

`-nulls` 可选的数值是一个空白字符串和字符 `'n'` ， 告诉 `spi_execp` 哪些参数是 NULL 。如果给出，那么它必需和 `_value-list_` 的长度相同。 如果没有给出，那么所有参数值都是非 NULL 。

除了查询及其参数声明的方式之外， `spi_execp` 的使用方法基本上和 `spi_exec` 一样。 `-count` , `-array` , `_loop-body_` 选项都是一样的，结果数值也一样。

下面是一个使用预备规划的 PL/Tcl 函数的例子：

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num &gt;= \$1 AND num &lt;= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

需要在给 `spi_prepare` 的查询字符串里放反斜杠，以确保 `` $``_n_ `` 标记会原样传递给 `spi_prepare` ， 而不是被 Tcl 的变量代换替换掉。

`spi_lastoid`

如果该查询是单行 `INSERT` 并且被修改的表包含 OID ， 则返回最后的 `spi_exec` 或 `spi_execp` 查询插入的行的 OID 。如果不是，将得到零。

`quote` `_string_`

在给出的字符串里将所由单引号和反斜杠字符复制成双份。它可以用于安全地处理那些要输入到 `spi_exec` 或 `spi_prepare` 中的 SQL 命令中的引号包围字符串。 比如，假如一个 SQL 命令看起来像这样:

```
"SELECT '$val' AS ret"
```

这里的 Tcl 变量 `val` 实际上包含 `doesn't`。 这样最后的命令字符串会是这样：

```
SELECT 'doesn't' AS ret
```

而这个字符串在 `spi_exec` 或 `spi_prepare` 的时候会导致一个分析错误。为了能工作正常，提交的命令应该包含：

```
SELECT 'doesn''t' AS ret
```

在 PL/Tcl 中可以这样构造：

```
"SELECT '[ quote $val ]' AS ret"
```

`spi_execp` 的一个优点是你不需要像这样引号包围参数值， 因为参数绝不会当做 SQL 查询字符串的一部分被分析。

`elog` `_level_` `_msg_`

发出一个日志或者错误消息。可能的级别是 `DEBUG`，`LOG`，`INFO`，`NOTICE`，`WARNING`，`ERROR` 和 `FATAL`。 `ERROR` 抛出一个错误条件：如果没有被周围的Tcl代码捕获，那么该错误传到调用的查询中， 导致当前事务或子事务退出。作用和Tcl的 `error` 命令相同。 `FATAL` 退出当前事务并且导致当前会话关闭(可能在 PL/Tcl 函数里没有什么理由使用这个错误级别， 提供它主要是为了完整)。其他级别只产生不同的优先级信息。 某个优先级别的信息是报告给客户端还是写到服务器日志，还是两个都做是由log_min_messages 和client_min_messages配置变量控制的。参阅Chapter 18获取更多细节。

41.6. PL/Tcl里的触发器过程

触发器过程可以用 PL/Tcl 写。PostgreSQL 要求当做触发器调用的过程必需声明为没有参数并且返回类型为 `trigger` 的函数。

触发器管理器传递给过程体的信息是通过下面变量传递的：

`$TG_name`

`CREATE TRIGGER` 语句里的触发器名称。

`$TG_relid`

导致触发器被调用的表的对象 ID 。

`$TG_table_name`

导致触发器被调用的表的名字。

`$TG_table_schema`

导致触发器被调用的表的模式。

`$TG_relatts`

以一个空表元素为前导的表中字段名称的 Tcl 列表。所以用Tcl命令 `lsearch` 在列表里查找元素名称时，返回的从 1 开始计数的正整数，与PostgreSQL 里字段编号的传统一样。已经被删除掉的字段位置的空的列表元素仍然会出现，这样， 属性编号与字段的对应就是正确的。

`$TG_when`

由触发器事件类型决定的字符串 `BEFORE` , `AFTER` 或 `INSTEAD OF`

`$TG_level`

由触发器事件类型决定的字符串 `ROW` 或 `STATEMENT`

`$TG_op`

由触发器事件类型决定的字符串 `INSERT` , `UPDATE` , `DELETE` 或 `TRUNCATE`

`$NEW`

一个关联数组，包含 `INSERT` 或 `UPDATE` 动作的新表行值， 如果是 `DELETE` 则为空。该数组是用字段名做索引的。那些为空的字段不会在数组中出现。 这不是为语句级别的触发器设置的。

`$OLD`

一个关联数组，包含 `UPDATE` 或 `DELETE` 动作的新表行， 如果是 `INSERT` 则为空。该数组是用字段名做索引的。那些为空的字段不会在数组中出现。 这不是为语句级别的触发器设置的。

`$args`

如同在 `CREATE TRIGGER` 语句里给出的参数一样的 Tcl 程序参数表。 这些参数在过程体里可以通过 `$1` ... `$``_n_` 来访问。

触发器过程返回的值是字符串 `OK` 或 `SKIP` 之一，或者一个像 `array get` Tcl 命令返回的数组那样的东西。如果返回值是 `OK`，触发触发器的操作 (`INSERT` / `UPDATE` / `DELETE`)将会正常进行。 `SKIP` 告诉触发器管理器不声不响地忽略该行的操作。如果返回一个数组， 那么它告诉 PL/Tcl 返回一个修改后的行给触发器管理器。这仅仅对行级别的 `BEFORE` `INSERT` 或 `UPDATE` 触发器有意义，修改的行而不是 `$NEW` 里面给出的行被插入； 或对于行级别的 `INSTEAD OF` `INSERT` 或 `UPDATE` 触发器， 返回的行用来提供给 `INSERT RETURNING` 或 `UPDATE RETURNING` 命令。 其他触发器类型忽略返回值。

下面是一个小的触发器过程的例子，它强制表内的一个整数值对行的更新次数进行跟踪。 对插入的新行，该值初始化为 0 并且在每次更新操作中加一。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

请注意触发器过程本身并不知道字段名字；那些是从触发器参数中提供的。 这样就可以让将触发器过程复用于不同的表。

41.7. 模块和 `unknown` 的命令

PL/Tcl 使用时支持自动加载 Tcl 代码。它识别一个特殊的 `pltcl_modules` 表， 该表被认为包含Tcl 代码的模块。如果存在这个表，则在数据库会话中的PL/Tcl函数第一次执行之前， 从该表中抓取 `unknown` 模块并加载到 Tcl 解释器中。（如果在一个会话中有多个， 将单独为每个Tcl解释器发生；参阅Section 41.4。）

因为 `unknown` 模块实际上可以包含任何你需要的初始化脚本，它通常是定义为一个Tcl `unknown` 过程，在 Tcl 不能识别一个调用的过程名的时候就调用它。PL/Tcl 这个过程的标准版本试图在 `pltcl_modules` 里找到一个定义所需要过程的模块。 如果找到一个，那么把它加载入解释器，然后允许继续按照原来的过程调用处理。另外还定义了一个 `pltcl_modfuncs` 表，它提供了哪个函数由哪个模块定义的索引，因此查找过程相当快。

PostgreSQL包括维护这些表的支持脚本：`pltcl_loadmod`，`pltcl_listmod`，`pltcl_delmod` 以及标准 `share/unknown.pltcl` 中 `unknown` 模块的源代码。这个模块必须一开始就加载入每个数据库才能支持自动加载机制。

表 `pltcl_modules` 和 `pltcl_modfuncs` 必需可以为所有人读取， 但是把它做成只有数据库管理员可写并拥有是明智的。作为一个安全预防， PL/Tcl将忽略 `pltcl_modules` （并且因此，不尝试加载 `unknown` 模块） 除非它被超级用户所有。但是这个表上的更新权限可以赋予给其他用户，如果你足够信任他们。

41.8. Tcl 过程名字

在PostgreSQL里，同一个函数名字可以用于不同的函数定义， 只要参数个数或者它们的类型不同。不过，Tcl 要求所有的过程名字都是唯一的。 PL/Tcl 通过把内部 Tcl 过程名字包含该函数来自系统表 `pg_proc` 的对象 ID 作为名字的一部分来处理这些问题。 因此同名不同参数类型的PostgreSQL函数也将会有不同的 Tcl 过程名。 这个问题通常对 PL/Tcl 程序员而言不算啥，但是在调试的时候可能会看到。

Chapter 42. PL/Perl - Perl 过程语言

Table of Contents

PL/Perl 是一种可加载的过程语言，通过它可以用Perl 编程语言 编写PostgreSQL函数。

使用 PL/Perl 的主要优点是允许在函数中大量使用来自 Perl 的处理字符串的操作和函数。PL/pgSQL 很难分析的复杂字符串对 Perl 来说却是小菜一碟。

要在特定数据库里安装 PL/Perl ，使用 `CREATE EXTENSION plperl` ，或在shell命令行里使用 `createlang plperl` `_dbname_` 。

> **Tip:** 如果某种编程语言安装到 `template1` ，那么所有随后创建的数据库都会自动安装这种语言。
>
> **Note:** 使用源码包的用户必须在安装过程中特别打开 PL/Perl 的编译。请参考Chapter 15 获取更多信息。二进制包的用户可能会在一些独立的子包中找到 PL/Perl 。

42.1. PL/Perl 函数和参数

要用 PL/Perl 语言创建一个函数，可以使用标准的CREATE FUNCTION语法：

```
CREATE FUNCTION _funcname_ (_argument-types_) RETURNS _return-type_ AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

函数体是普通 Perl 代码。实际上，PL/Perl 胶水代码将其封装在一个 Perl 子过程里。 一个 PL/Perl 函数在一个标量环境中调用，所以不能返回一个列表。 你可以像下面描述的那样用返回引用的方法返回非标量值（arrays, records, 和 sets）。

PL/Perl也支持DO语句调用匿名代码块：

```
DO $$
    # PL/Perl code
$$ LANGUAGE plperl;
```

一个匿名代码块不接收参数，而且丢弃任何返回值。否则它的行为就像一个函数。

> **Note:** 在 Perl 里使用命名的嵌套子过程是很危险的，特别是它们在闭包里引用了词法变量的时候。 因为 PL/Perl 是封装在一个子过程里，因此，任何你放进去的命名子过程都将被嵌套。通常， 创建一个用 coderef 调用的匿名子过程要安全得多。想要获取更多细节，请参阅 `Variable "%s" will not stay shared` 里的记录或 perldiag手册页中的 `Variable "%s" is not available` ， 或在Internet上搜索"perl 嵌套命名子过程"。

`CREATE FUNCTION` 命令的语法要求把函数体写成字符串常量。 通常处理字符串文本用美元符界定更方便(参阅Section 4.1.2.4)， 如果你想使用传统的 `E''` 逃逸语法，必须双写函数体里使用的任何单引号(`'`) 和反斜杠(`\`)(参见Section 4.1.2.1)。

参数和结果都是和任何其它 Perl 子过程里那样处理的：参数是放在 `@_` 里传递的， 结果值是用 `return` 返回或者作为函数中最后计算的表达式的值返回。

比如，一个返回两个整数中较大值的函数可以这么写：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

> **Note:** 为了能够在PL/Perl里使用，参数将从数据库编码转换为UTF-8，然后在返回时从UTF-8回到数据库编码。

如果给函数传递一个 NULL 那么其参数值将以 Perl 中"undefined" 的形式出现。上面的函数定义在输入为 NULL 时的行为不是很正常(实际上， 它将表现得好像它们都是零一样)。可以给函数定义增加 `STRICT` 让 PostgreSQL做一些更合理的事情：如果传递进来一个 NULL， 那么该函数则根本不会被调用，而只是自动返回一个 NULL 结果。另外， 可以在函数体里检查未定义的输入。比如，假设想收到一个 NULL 和一个非 NULL 参数的 `perl_max` 返回非 NULL 的参数，而不是 NULL ：

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

如上所述，要从 PL/Perl 函数中返回一个 NULL，可以返回一个未定义的数值。 不管该函数是否严格，都可以这么做。

任何一个不是引用的函数参数是一个字符串，这是相关数据类型在标准PostgreSQL 外部文本里的表示。普通数字或文本类型的情况下，Perl将只是做正确的事情，程序员不需要担心。然而，在其他情况下，需要将参数转换为Perl可用的形式。例如， `decode_bytea` 函数可以用来转换类型 `bytea` 的参数为非逃逸的二进制。

相似的，传回PostgreSQL的值必须是外部文本表示格式。 例如， `encode_bytea` 函数可以用来为一个类型为 `bytea` 的返回值逃逸二进制数据。

Perl 可以用 Perl 数组引用的方式返回PostgreSQL数组。 下面是一个例子：

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl作为一个 `PostgreSQL::InServer::ARRAY` 对象传递PostgreSQL 数组。这个对象可以被视为一个数组引用或一个字符串，Perl为PostgreSQL 低于9.1的版本编写了代码，允许向后兼容。例如：

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # as an array reference
    for (@$arg) {
        $result .= $_;
    }

    # also works as a string
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **Note:** 多维数组的表现就和每个Perl程序员引用低维数组的引用一样。

复合类型的参数是当做指向散列的引用传递给函数的。散列的键字是复合类型的属性名。下面是一个例子：

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

使用同样的办法，一个 PL/Perl 函数可以返回一个复合类型的结果： 返回一个包含所需要的属性的散列的引用。比如，

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

在声明的结果数据类型里的任何字段如果在散列里面没有出现，那么都会当作 NULL 返回。

PL/Perl 函数也能返回标量或者复合类型的集合。通常你希望一次返回一行， 一方面加速函数启动时间，另外一方面防止在内存里堆积整个结果集。 可以用下面说明的函数 `return_next` 。请注意在最后的 `return_next` ，你必须放一个 `return` 或者(最好是) `return undef` 。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

对于小的结果集，你可以返回一个指向一个数组的引用，这个数组可以包含标量，指向数组的引用， 或者指向简单类型，数组类型以及复合类型等的散列的引用。这里是一个简单的例子， 它把整个结果集当作一个数组引用返回：

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

如果你想在自己的代码里使用 `strict` 用法你有几种选择。对于临时全局使用你可以 `SET plperl.use_strict` 为真。这个参数影响随后的PL/Perl 函数的编译，但是不影响在当前会话里已经编译了的函数。为了永久全局使用， 可以在 `postgresql.conf` 文件里设置 `plperl.use_strict` 为真。

要在特定的函数中永久使用，只需要简单地在函数体的顶部放置：

```
use strict;
```

如果你的Perl是版本5.10.0或更高，那么 `feature` 程序也适用于 `use` 。

42.2. PL/Perl里的数据值

提供给 PL/Perl 函数代码的参数值只是简单地将输入参数转换成文本形式(就像它们被 `SELECT` 语句显示的那样)。与之相对的是， `return` 和 `return_next` 命令将接受任何函数声明的返回类型可以接受的输入格式的字符串。

42.3. 内置函数

42.3.1. 从PL/Perl访问数据库

从 Perl 函数里访问数据库本身可以通过下面的函数做到：

`spi_exec_query`(` *query* `[,` *max-rows*`])

`spi_exec_query` 执行一个 SQL 命令然后把整个结果集当作一个指向散列引用的引用返回。只有在你知道结果集相对比较小的时候才能用这个命令。 下面是一个带有额外的最大行数的查询(`SELECT` 命令)的例子。

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

它从 `my_table` 里返回最多 5 行。如果 `my_table` 有一个字段是 `my_column` ，那么可以用下面的方法从结果的第 `$i` 行获取其值：

```
$foo = $rv-&gt;{rows}[$i]-&gt;{my_column};
```

从一个 `SELECT` 查询返回的总行数可以这样访问：

```
$nrows = $rv-&gt;{processed}
```

这里是一个使用其它命令的例子：

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

你可以用下面方法访问状态(如 `SPI_OK_INSERT`)：

```
$res = $rv-&gt;{status};
```

这样获取影响的行数：

```
$nrows = $rv-&gt;{processed};
```

下面是一个完整的例子：

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv-&gt;{status};
    my $nrows = $rv-&gt;{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv-&gt;{rows}[$rn];
        $row-&gt;{i} += 200 if defined($row-&gt;{i});
        $row-&gt;{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row-&gt;{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query(```_command_`)` spi*fetchrow(```_cursor* `)` ``spi_cursor_close(```_cursor_)

`spi_query` 和 `spi_fetchrow` 一起用于处理那些行集可能比较大， 或者是在你收到行的时候就返回的场合。 `spi_fetchrow` 只能 和 `spi_query` 一起使用。下面的例子演示了如何使用：

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '&lt;', $file # 这是访问文件！
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = &lt;$fh&gt;;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num =&gt; $row-&gt;{a},
            the_text =&gt; md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

通常，应当重复 `spi_fetchrow` 直到其返回 `undef` 以表示没有行可以读取了，此时由 `spi_query` 返回的游标将被自动释放。 如果你确实不想读取所有行，可以明确调用 `spi_cursor_close` 来释放游标， 否则将会导致内存泄漏。

`spi_prepare(```_command_`, `_argument types_`)` spi*query_prepared(```_plan* , *arguments*) [, `_attributes_`], `_arguments_`) `` `spi_freeplan(```_plan_ ``)

`spi_prepare` , `spi_query_prepared` , `spi_exec_prepared` , `spi_freeplan` 为预备查询实现同样的功能。 `spi_prepare` 接受一个带有编号的参数占位符的字符串和一个参数类型的字符串列表：

```
$plan = spi_prepare('SELECT * FROM test WHERE id &gt; $1 AND name = $2',
                                                     'INTEGER', 'TEXT');
```

一旦一个查询规划通过调用 `spi_prepare` 准备好，该规划就可以代替查询字符串， 不管是在 `spi_exec_prepared` 中(与 `spi_exec_query` 返回的结果相同) 还是在 `spi_query_prepared` 中 (与 `spi_query` 返回的游标相同)， 之后可以被传递给 `spi_query` 。 `spi_exec_prepared` 可选的第二个参数是一个属性的哈希引用；当前唯一支持的属性是 `limit` ， 设置查询返回行的最小数量。

预备查询的好处是可以为多个查询的执行使用一个预备规划。在规划不再被需要之后， 可以通过 `spi_freeplan` 释放：

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )-&gt;{rows}-&gt;[0]-&gt;{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

注意， `spi_prepare` 中的参数是通过 $1, $2, $3 ... 表示的， 因此避免在双引号中声明查询字符串，那样可能会导致难以发现的臭虫。

另外一个说明 `spi_exec_prepared` 里的可选参数的使用的例子：

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address &lt;&lt; $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit =&gt; 2},
                $_[0]
        )-&gt;{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

42.3.2. PL/Perl里的效用函数

`elog(```_level_ , _msg_)

发出一条日志或者错误信息。可能的级别是 `DEBUG` , `LOG` , `INFO` , `NOTICE` , `WARNING` , `ERROR` 。 `ERROR` 抛出一个错误条件。 如果这个错误没有被周围的 Perl 代码捕获，那么错误将传播到调用的查询里， 导致当前事务或者子事务退出。这实际上相当于 Perl 的 `die` 命令。其它级别只是生成不同优先级的消息。特定优先级的消息是否报告给客户端、写到服务器日志、 或者两个都做，是由配置参数log_min_messages和client_min_messages 控制的。参阅Chapter 18获取更多信息。

`quote_literal(```_string_)

适当的返回在一个SQL语句字符串中作为字符串文本引用的给定字符串。嵌入的单引号和反斜杠要加一倍。 请注意， `quote_literal` 在未定义的输入上返回未定义；如果参数是未定义的， `quote_nullable` 往往是更合适的。

`quote_nullable(```_string_)

适当的返回在一个SQL语句字符串中作为字符串文本引用的给定字符串。或者，如果参数是未定义的， 返回不加引号的字符串"NULL"。嵌入的单引号和反斜杠要加一倍。

`quote_ident(```_string_)

适当的返回在一个SQL语句字符串中作为一个标识符引用的给定字符串。只有在必要时添加引号（也就是，如果字符串包含非标识符字符或是case-folded）。嵌入的单引号和反斜杠要加一倍。

`` `decode_bytea(```_string_ ``)

返回通过给定字符串内容表示的非逃逸二进制数据，应该是 `bytea` 编码。

`` `encode_bytea(```_string_ ``)

返回给定字符串的二进制数据内容的 `bytea` 编码格式。

`` encode_array_literal(```_array_`) `` encode*array_literal(```_array* , *delimiter*`)

返回引用的数组的内容，以在数组里的字符串文本的格式(参阅Section 8.15.2)。 如果不是一个数组的引用则返回未改变的参数值。如果分隔符没用指定或是未定义的， 则数组文字元素之间的分隔符缺省是" `,` "。

`` `encode_typed_literal(```_value_ ``, `` _typename_ ``)

转换一个Perl变量为作为第二个参数传递的数据类型的值和返回一个这个值的字符串表示。正确处理嵌套数组和复合类型的值。

`` `encode_array_constructor(```_array_ ``)

返回引用的数组的内容，以在数组构造器里的字符串的格式(参阅Section 4.2.12)。 个别的值用 `quote_nullable` 引用。如果不是对数组的引用，那么返回参数值用 `quote_nullable` 引用。

`` `looks_like_number(```_string_ ``)

根据Perl，如果给定字符串的内容看起来像一个数字则返回真，否则返回假。如果参数是未定义则返回未定义。 忽略前置和后置的空格。 `Inf` 和 `Infinity` 被认为是数字。

`` `is_array_ref(```_argument_ ``)

如果给定参数可能被视为一个数组引用则返回真，也就是，如果参数的参考是 `ARRAY` 或 `PostgreSQL::InServer::ARRAY` 。否则返回假。

42.4. PL/Perl里的全局变量

你可以利用全局散列 `%_SHARED` 保存数据，包括代码引用， 持续时间可以达到当前会话的生命期。

下面是一个共享数据的例子：

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

下面是一个使用代码引用的稍微复杂些的例子：

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* 初始化函数 */

/* 建立一个使用引用的函数的函数 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

如果不在乎易读性，你可以用一行 `return $_SHARED{myquote}->($_[0]);` 代替上面的三行。

为了安全原因，PL/Perl通过一个SQL角色在一个单独为这个角色的解释器里执行函数调用。 这阻止了有另一个用户的PL/Perl函数的表现的用户的恶意干扰或事故。每个这样的解释器有它自己的 `%_SHARED` 变量值和其他全局状态。因此，当且仅当通过相同的SQL角色执行时， 两个PL/Perl函数才将分享相同的 `%_SHARED` 值。在多个SQL角色下， 只有一个会话执行代码的应用中（通过 `SECURITY DEFINER` 函数，使用 `SET ROLE` 等）， 可能需要采取明确的步骤保证PL/Perl函数可以通过 `%_SHARED` 分享数据。 要做到这点，确保应该沟通的函数是由相同的用户所有的，并且标记他们为 `SECURITY DEFINER` 。 当然必须要注意这样的函数不能用来做意料之外的事情。

42.5. 可信的和不可信的 PL/Perl

通常，PL/Perl 是作为一种叫 `plperl` 的"可信"编程语言安装的。在这种设置中， 为了保持安全，某些 Perl 操作被关闭掉了。通常，受限制的操作都是那些和环境相互交互的动作。 这包括文件句柄操作、 `require` 、 `use` (对于外部模块)。 没有办法访问数据库服务器进程内部或者获取具有服务器进程权限的 OS 级别的访问，就像 C 函数那样。 因此，任何非特权的数据库用户都可以允许使用这种语言。

这里是一个无法运转的函数的例子，因为出于安全原因，文件系统的操作是不允许的：

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

创建这个函数将会失败，因为它使用的非法调用将会被验证器捕获。

有时候想写不受限制的 Perl 函数。比如，可能需要一个能发送邮件的 Perl 函数。为了处理这种情况， PL/Perl 也可以安装为"不可信的"的语言PL/PerlU。 在这种情况下，可以使用完整的 Perl 语言。当安装这个语言时， `plperlu` 这个名字可以选取不可信的 PL/Perl 变种。

PL/PerlU函数的作者必须注意不能把该函数用于做任何不想做的事情， 因为它可以干任何数据库管理员能干的事情。请注意数据库系统只允许数据库超级用户创建不可信语言写的函数。

如果上面的函数由超级用户用 `plperlu` 创建，那么执行就会成功。

同样的方式，如果语言被声明为 `plperlu` 而不是 `plperl` ， 那么用Perl写的匿名代码块可以使用受限制的操作，但是调用者必须是超级用户。

> **Note:** 当PL/Perl函数在一个单独的为每个SQL用户的Perl解释器里运行时， 所有的PL/PerlU函数在一个单独的Perl解释器里的给定会话中执行 （不是用于PL/Perl函数的那个解释器）。这允许PL/PerlU 函数自由的分享数据，但是在PL/Perl和PL/PerlU 函数之间不会有通讯发生。
>
> **Note:** Perl在一个进程中不能支持多个解释器，除非在编译时给它适当的标志， `usemultiplicity` 或 `useithreads` 。 （ `usemultiplicity` 是首选的，除非你实际需要使用线程。 获取更多信息，请参阅perlembed手册页。） 如果PL/Perl与一个没有这样编译的Perl的副本一起使用，那么每个会话只可能有一个Perl解释器， 所以任意一个会话只能执行PL/PerlU函数，或PL/Perl函数， 他们都是通过同一个SQL角色调用的。

42.6. PL/Perl 触发器

PL/Perl 可以用来书写触发器函数。在一个触发器函数中，散列引用 `$_TD` 包含了当前触发事件的信息。 `$_TD` 是一个全局变量， 它对于每次触发器调用都能够获取一个局部值。 `$_TD` 散列引用的字段有：

`$_TD->{new}{foo}`

字段 `foo` 的 `NEW` 值

`$_TD->{old}{foo}`

字段 `foo` 的 `OLD` 值

`$_TD->{name}`

被调用的触发器的名字

`$_TD->{event}`

触发器事件： `INSERT` , `UPDATE` , `DELETE` , `TRUNCATE` , 或 `UNKNOWN`

`$_TD->{when}`

何时调用触发器： `BEFORE` , `AFTER` , `INSTEAD OF` , 或 `UNKNOWN`

`$_TD->{level}`

触发器的级别： `ROW` , `STATEMENT` , 或 `UNKNOWN`

`$_TD->{relid}`

触发触发器的表的 OID

`$_TD->{table_name}`

触发触发器的表的名字

`$_TD->{relname}`

触发触发器的表的名字。已经废弃了，并且可能在将来的版本中移除。请使用$_TD->{table_name}。

`$_TD->{table_schema}`

触发触发器的表的模式名

`$_TD->{argc}`

触发器函数的参数个数

`@{$_TD->{args}}`

触发器函数的参数，如果 `$_TD->{argc}` 为 0 则不存在。

行级别的触发器返回下列之一：

`return;`

执行该操作

`"SKIP"`

不执行该操作

`"MODIFY"`

表明 `NEW` 行被触发器函数修改过

下面是一个触发器函数，演示了上面的一些东西。

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    #跳过 INSERT/UPDATE 命令
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  #修改一行并且执行 INSERT/UPDATE 命令

    } else {
        return;           # 执行 INSERT/UPDATE 命令
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

42.7. 后台PL/Perl

42.7.1. 配置

本节列出影响PL/Perl的配置参数。

`plperl.on_init` (`string`)

指定当Perl触发器第一次初始化时执行Perl代码，在这之前是专业为了 `plperl` 或 `plperlu` 使用的。当这段代码执行时SPI函数是不适用的。如果代码因错误而失败， 那么它将退出解释器的初始化，并且传播错误到调用的查询，导致当前事务或子事务退出。

Perl代码限制为单个字符串。更长的代码可以放入一个模块，并且由 `on_init` 字符串加载。例子：

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

任何由 `plperl.on_init` 直接或非直接加载的模块，都将适用于 `plperl` 使用。 这可能会创建一个安全风险。要查看哪个模块被加载了可以使用：

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

如果plperl库包含在shared_preload_libraries里面，那么初始化将在postmaster中发生， 这种情况下要额外考虑postmaster不稳定的风险。利用这一特性的首要原因是Perl模块通过 `plperl.on_init` 加载需要在postmaster启动的情况下，并且在个人的数据库会话中不会有额外开销加载，立即可用。然而， 请记住，额外开销只是在第一次Perl解释器被数据库会话使用的时候避免， 也就是PL/PerlU或PL/Perl是调用PL/Perl函数的第一个SQL角色。 在一个数据库会话中创建的任何额外的Perl解释器必须重新执行 `plperl.on_init` 。同样，在Windows上， 从预加载中不会有任何节省，因为在postmaster进程中创建的Perl解释器不会传播到子进程。

这个postmaster只能在 `postgresql.conf` 文件或服务器命令行中设置。

`plperl.on_plperl_init` (`string`) `plperl.on_plperlu_init` (`string`)

这些参数指定当Perl解释器专门分别为 `plperl` 或 `plperlu` 时执行Perl代码。 当PL/Perl或PL/PerlU函数在一个数据库会话中第一次执行时会发生这种情况， 或当一个额外的解释器因为其他语言的调用或一个PL/Perl函数被一个新的SQL角色调用而创建时， 也会发生这种情

况。这遵循任何由`plperl.on_init`所做的初始化。 当执行Perl代码时无法使用SPI函数。在`plperl.on_plperl_init`里面的Perl代码在 "锁定"解释器后执行，并且因此只能执行信任的操作。

如果代码因错误而失败，那么它将退出解释器的初始化，并且传播错误到调用的查询，导致当前事务或子事务退出。 任何Perl已经做的动作不会回滚；但是，那个解释器将不会再使用。如果语言再次被使用， 那么初始化将在一个新的Perl解释器中尝试。

只有超级用户可以改变这些设置。尽管这些设置可以在一个会话中改变， 但是这些改变将不会影响到已经用来执行函数的Perl解释器。

`plperl.use_strict` (`boolean`)

当设置PL/Perl函数的真实的后续编译时，将启用`strict`编译指示。 这个参数不影响已经用当前会话编译过的函数。

42.7.2. 限制及缺少的特性

下面的特性目前还在 PL/Perl 里面缺少，但是欢迎贡献。

- PL/Perl 函数无法相互直接调用。
- SPI 目前尚未完全实现。
- 如果你用 `spi_exec_query` 抓取非常大的数据集，你应该明白这些数据都会放到内存里。你可以用前面演示的 `spi_query` / `spi_fetchrow` 来避免这些。

  类似的问题也会发生在一个返回集合的函数用 `return` 给 PostgreSQL 传递回去一个巨大的行集合。你也可以像前面演示的那样用 `return_next` 返回每个返回行的方法避免这个问题。
- 当一个会话正常结束时，不是因为一个致命的错误而结束，执行任何已经定义的 `END` 块。 当前没有其他动作执行。特别的，文件句柄不自动刷新并且对象不自动销毁。

Chapter 43. PL/Python - Python 过程语言

Table of Contents

PL/Python 过程语言允许用Python语言编写 PostgreSQL 函数。 Python 语言.

要在特定的数据库里安装PL/Python，使用 `CREATE EXTENSION plpythonu` ，或者使用命令行工具 `createlang plpythonu` _数据库名称_ （详见Section 43.1）。

> **Tip:** 如果一门语言安装到了 `template1` 数据库中，那么所有的随后创建的数据库都会自动的安装该语言。

到目前为止PostgreSQL ， PL/Python 只能当成 一种"不可信任的" 语言，意思是他没有提供任何限制用户可为与不可为的手段，因此 他被重命令为 `plpythonu` ，可信任的 `plpython` 可能在将来某个时间能够获得 if a new secure execution mechanism is developed in Python. The writer of a function in untrusted PL/Python must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Only superusers can create functions in untrusted languages such as `plpythonu` .

> **Note:** Users of source packages must specially enable the build of PL/Python during the installation process. (Refer to the installation instructions for more information.) Users of binary packages might find PL/Python in a separate subpackage.

43.1. Python 2 vs. Python 3

PL/Python supports both the Python 2 and Python 3 language variants. (The PostgreSQL installation instructions might contain more precise information about the exact supported minor versions of Python.) Because the Python 2 and Python 3 language variants are incompatible in some important aspects, the following naming and transitioning scheme is used by PL/Python to avoid mixing them:

- The PostgreSQL language named `plpython2u` implements PL/Python based on the Python 2 language variant.
- The PostgreSQL language named `plpython3u` implements PL/Python based on the Python 3 language variant.
- The language named `plpythonu` implements PL/Python based on the default Python language variant, which is currently Python 2. (This default is independent of what any local Python installations might consider to be their "default", for example, what `/usr/bin/python` might be.) The default will probably be changed to Python 3 in a distant future release of PostgreSQL, depending on the progress of the migration to Python 3 in the Python community.

This scheme is analogous to the recommendations in PEP 394 regarding the naming and transitioning of the `python` command.

It depends on the build configuration or the installed packages whether PL/Python for Python 2 or Python 3 or both are available.

> **Tip:** The built variant depends on which Python version was found during the installation or which version was explicitly set using the `PYTHON` environment variable; see Section 15.4. To make both variants of PL/Python available in one installation, the source tree has to be configured and built twice.

This results in the following usage and migration strategy:

- Existing users and users who are currently not interested in Python 3 use the language name `plpythonu` and don't have to change anything for the foreseeable future. It is recommended to gradually "future-proof" the code via migration to Python 2.6/2.7 to simplify the eventual migration to Python 3.

  In practice, many PL/Python functions will migrate to Python 3 with few or no changes.

- Users who know that they have heavily Python 2 dependent code and don't plan to ever change it can make use of the `plpython2u` language name. This will continue to work into the very distant future, until Python 2 support might be completely dropped by PostgreSQL.
- Users who want to dive into Python 3 can use the `plpython3u` language name, which will keep working forever by today's standards. In the distant future, when Python 3 might become the default, they might like to remove the "3" for aesthetic reasons.
- Daredevils, who want to build a Python-3-only operating system environment, can change the contents of `pg_pltemplate` to make `plpythonu` be equivalent to `plpython3u`, keeping in mind that this would make their installation incompatible with most of the rest of the world.

See also the document What's New In Python 3.0 for more information about porting to Python 3.

It is not allowed to use PL/Python based on Python 2 and PL/Python based on Python 3 in the same session, because the symbols in the dynamic modules would clash, which could result in crashes of the PostgreSQL server process. There is a check that prevents mixing Python major versions in a session, which will abort the session if a mismatch is detected. It is possible, however, to use both PL/Python variants in the same database, from separate sessions.

43.2. PL/Python Functions

Functions in PL/Python are declared via the standard CREATE FUNCTION syntax:

```
CREATE FUNCTION _funcname_ (_argument-list_)
  RETURNS _return-type_
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

The body of a function is simply a Python script. When the function is called, its arguments are passed as elements of the list `args` ; named arguments are also passed as ordinary variables to the Python script. Use of named arguments is usually more readable. The result is returned from the Python code in the usual way, with `return` or `yield` (in case of a result-set statement). If you do not provide a return value, Python returns the default `None` . PL/Python translates Python's `None` into the SQL null value.

For example, a function to return the greater of two integers can be defined as:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

The Python code that is given as the body of the function definition is transformed into a Python function. For example, the above results in:

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

assuming that 23456 is the OID assigned to the function by PostgreSQL.

The arguments are set as global variables. Because of the scoping rules of Python, this has the subtle consequence that an argument variable cannot be reassigned inside the function to the value of an expression that involves the variable name itself, unless the variable is redeclared as global in the block. For example, the following won't work:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpythonu;
```

because assigning to `x` makes `x` a local variable for the entire block, and so the `x` on the right-hand side of the assignment refers to a not-yet-assigned local variable `x`, not the PL/Python function parameter. Using the `global` statement, this can be made to work:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
$$ LANGUAGE plpythonu;
```

But it is advisable not to rely on this implementation detail of PL/Python. It is better to treat the function parameters as read-only.

43.3. Data Values

Generally speaking, the aim of PL/Python is to provide a "natural" mapping between the PostgreSQL and the Python worlds. This informs the data mapping rules described below.

43.3.1. Data Type Mapping

Function arguments are converted from their PostgreSQL type to a corresponding Python type:

- PostgreSQL `boolean` is converted to Python `bool`.
- PostgreSQL `smallint` and `int` are converted to Python `int`. PostgreSQL `bigint` and `oid` are converted to `long` in Python 2 and to `int` in Python 3.
- PostgreSQL `real`, `double`, and `numeric` are converted to Python `float`. Note that for the `numeric` this loses information and can lead to incorrect results. This might be fixed in a future release.
- PostgreSQL `bytea` is converted to Python `str` in Python 2 and to `bytes` in Python 3. In Python 2, the string should be treated as a byte sequence without any character encoding.
- All other data types, including the PostgreSQL character string types, are converted to a Python `str`. In Python 2, this string will be in the PostgreSQL server encoding; in Python 3, it will be a Unicode string like all strings.
- For nonscalar data types, see below.

Function return values are converted to the declared PostgreSQL return data type as follows:

- When the PostgreSQL return type is `boolean`, the return value will be evaluated for truth according to the *Python* rules. That is, 0 and empty string are false, but notably `'f'` is true.
- When the PostgreSQL return type is `bytea`, the return value will be converted to a string (Python 2) or bytes (Python 3) using the respective Python built-ins, with the result being converted `bytea`.
- For all other PostgreSQL return types, the returned Python value is converted to a string using the Python built-in `str`, and the result is passed to the input function of the PostgreSQL data type.

Strings in Python 2 are required to be in the PostgreSQL server encoding when they are passed to PostgreSQL. Strings that are not valid in the current server encoding will raise an error, but not all encoding mismatches can be detected, so garbage data can still result when this is not done correctly. Unicode strings are converted to the correct encoding automatically, so it can be safer and more convenient to use those. In Python 3, all strings are Unicode strings.

- For nonscalar data types, see below.

Note that logical mismatches between the declared PostgreSQL return type and the Python data type of the actual return object are not flagged; the value will be converted in any case.

43.3.2. Null, None

If an SQL null value is passed to a function, the argument value will appear as `None` in Python. For example, the function definition of `pymax` shown in Section 43.2 will return the wrong answer for null inputs. We could add `STRICT` to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for null inputs in the function body:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

As shown above, to return an SQL null value from a PL/Python function, return the value `None`. This can be done whether the function is strict or not.

43.3.3. Arrays, Lists

SQL array values are passed into PL/Python as a Python list. To return an SQL array value out of a PL/Python function, return a Python sequence, for example a list or tuple:

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Note that in Python, strings are sequences, which can have undesirable effects that might be familiar to Python programmers:

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

43.3.4. Composite Types

Composite-type arguments are passed to the function as Python mappings. The element names of the mapping are the attribute names of the composite type. If an attribute in the passed row has the null value, it has the value `None` in the mapping. Here is an example:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

There are multiple ways to return row or composite types from a Python function. The following examples assume we have:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

A composite result can be returned as a:

Sequence type (a tuple or list, but not a set because it is not indexable)

Returned sequence objects must have the same number of items as the composite result type has fields. The item with index 0 is assigned to the first field of the composite type, 1 to the second and so on. For example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return [ name, value ]
  # or alternatively, as tuple: return ( name, value )
$$ LANGUAGE plpythonu;
```

To return a SQL null for any column, insert `None` at the corresponding position.

Mapping (dictionary)

The value for each result type column is retrieved from the mapping with the column name as key. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

Any extra dictionary key/value pairs are ignored. Missing keys are treated as errors. To return a SQL null value for any column, insert `None` with the corresponding column name as the key.

Object (any object providing method `__getattr__`)

This works the same as a mapping. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

Functions with `OUT` parameters are also supported. For example:

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

43.3.5. Set-returning Functions

A PL/Python function can also return sets of scalar or composite types. There are several ways to achieve this because the returned object is internally turned into an iterator. The following examples assume we have composite type:

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

A set result can be returned from a:

Sequence type (tuple, list, set)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

Iterator (any object providing `__iter__` and `next` methods)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Generator (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

Warning

Due to Python bug #1483133, some debug versions of Python 2.4 (configured and compiled with option `--with-pydebug`) are known to crash the PostgreSQL server when using an iterator to return a set result. Unpatched versions of Fedora 4 contain this bug. It does not happen in production versions of Python or on patched versions of Fedora 4.

Set-returning functions with `OUT` parameters (using `RETURNS SETOF record`) are also supported. For example:

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

43.4. Sharing Data

The global dictionary `SD` is available to store data between function calls. This variable is private static data. The global dictionary `GD` is public data, available to all Python functions within a session. Use with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the `GD` dictionary, as mentioned above.

43.5. Anonymous Code Blocks

PL/Python also supports anonymous code blocks called with the DO statement:

```
DO $$
    # PL/Python code
$$ LANGUAGE plpythonu;
```

An anonymous code block receives no arguments, and whatever value it might return is discarded. Otherwise it behaves just like a function.

43.6. Trigger Functions

When a function is used as a trigger, the dictionary `TD` contains trigger-related values:

`TD["event"]`

contains the event as a string: `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`.

`TD["when"]`

contains one of `BEFORE`, `AFTER`, or `INSTEAD OF`.

`TD["level"]`

contains `ROW` or `STATEMENT`.

`TD["new"]``TD["old"]`

For a row-level trigger, one or both of these fields contain the respective trigger rows, depending on the trigger event.

`TD["name"]`

contains the trigger name.

`TD["table_name"]`

contains the name of the table on which the trigger occurred.

`TD["table_schema"]`

contains the schema of the table on which the trigger occurred.

`TD["relid"]`

contains the OID of the table on which the trigger occurred.

`TD["args"]`

If the `CREATE TRIGGER` command included arguments, they are available in `TD["args"][0]` to `TD["args"][``_n_`-1].

If `TD["when"]` is `BEFORE` or `INSTEAD OF` and `TD["level"]` is `ROW`, you can return `None` or `"OK"` from the Python function to indicate the row is unmodified, `"SKIP"` to abort the event, or if `TD["event"]` is `INSERT` or `UPDATE` you can return `"MODIFY"` to indicate you've modified the new row. Otherwise the return value is ignored.

43.7. Database Access

The PL/Python language module automatically imports a Python module called `plpy`. The functions and constants in this module are available to you in the Python code as `plpy.``_foo_`.

43.7.1. Database Access Functions

The `plpy` module provides several functions to execute database commands:

`plpy.` execute `(``_query_` [, `_max-rows_`])

Calling `plpy.execute` with a query string and an optional row limit argument causes that query to be run and the result to be returned in a result object.

The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. For example:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as:

```
foo = rv[i]["my_column"]
```

The number of rows returned can be obtained using the built-in `len` function.

The result object has these additional methods:

`` `nrows `` ()`

Returns the number of rows processed by the command. Note that this is not necessarily the same as the number of rows returned. For example, an `UPDATE` command will set this value but won't return any rows (unless `RETURNING` is used).

`` `status `` ()`

The `SPI_execute()` return value.

``` colnames`()```coltypes`()` ``` coltypmods `()`

Return a list of column names, list of column type OIDs, and list of type-specific type modifiers for the columns, respectively.
```

These methods raise an exception when called on a result object from a command that did not produce a result set, e.g., `UPDATE` without `RETURNING`, or `DROP TABLE`. But it is OK to use these methods on a result set containing zero rows.

`` `__str__ ``()`

The standard `__str__` method is defined so that it is possible for example to debug query execution results using `plpy.debug(rv)`.

The result object can be modified.

Note that calling `plpy.execute` will cause the entire result set to be read into memory. Only use that function when you are sure that the result set will be relatively small. If you don't want to risk excessive memory usage when fetching large results, use `plpy.cursor` rather than `plpy.execute`.

`plpy.` prepare `` (``_query_ `` [, `_argtypes_` ]) `plpy.` execute `` (``_plan_ `` [, `_arguments_` [, `_max-rows_`]])

`plpy.prepare` prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

`text` is the type of the variable you will be passing for `$1`. The second argument is optional if you don't want to pass any parameters to the query.

After preparing a statement, you use a variant of the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, ["name"], 5)
```

Pass the plan as the first argument (instead of the query string), and a list of values to substitute into the query as the second argument. The second argument is optional if the query does not expect any parameters. The third argument is the optional row limit as before.

Query parameters and result row fields are converted between PostgreSQL and Python data types as described in Section 43.3. The exception is that composite types are currently not supported: They will be rejected as query parameters and are converted to strings when appearing in a query result. As a workaround for the latter problem, the query can sometimes be rewritten so that the composite type result appears as a result row rather than as a field of the result row. Alternatively, the resulting string could be parsed apart by hand, but this approach is not recommended because it is not future-proof.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Chapter 44) for a description of what this means. In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries `SD` or `GD` (see Section 43.4). For example:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    plan = SD.setdefault("plan", plpy.prepare("SELECT 1"))
    # rest of function
$$ LANGUAGE plpythonu;
```

plpy. cursor (``_query_ ) plpy. cursor (``_plan_ [, _arguments_])

The `plpy.cursor` function accepts the same arguments as `plpy.execute` (except for the row limit) and returns a cursor object, which allows you to process large result sets in smaller chunks. As with `plpy.execute`, either a query string or a plan object along with a list of arguments can be used.

The cursor object provides a `fetch` method that accepts an integer parameter and returns a result object. Each time you call `fetch`, the returned object will contain the next batch of rows, never larger than the parameter value. Once all rows are exhausted, `fetch` starts returning an empty result object. Cursor objects also provide an iterator interface, yielding one row at a time until all rows are exhausted. Data fetched that way is not returned as result objects, but rather as dictionaries, each dictionary corresponding to a single result row.

An example of two ways of processing data from a large table is:

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 &lt;&gt; 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

Cursors are automatically disposed of. But if you want to explicitly release all resources held by a cursor, use the `close` method. Once closed, a cursor cannot be fetched from anymore.

> **Tip:** Do not confuse objects created by `plpy.cursor` with DB-API cursors as defined by the Python Database API specification. They don't have anything in common except for the name.

43.7.2. Trapping Errors

Functions accessing the database might encounter errors, which will cause them to abort and raise an exception. Both `plpy.execute` and `plpy.prepare` can raise an instance of a subclass of `plpy.SPIError`, which by default will terminate the function. This error can be handled just like any other Python exception, by using the `try/except` construct. For example:

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

The actual class of the exception being raised corresponds to the specific condition that caused the error. Refer to Table A-1 for a list of possible conditions. The module `plpy.spiexceptions` defines an exception class for each PostgreSQL condition, deriving their names from the condition name. For instance, `division_by_zero` becomes `DivisionByZero`, `unique_violation` becomes `UniqueViolation`, `fdw_error` becomes `FdwError`, and so on. Each of these exception classes inherits from `SPIError`. This separation makes it easier to handle specific errors, for instance:

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

Note that because all exceptions from the `plpy.spiexceptions` module inherit from `SPIError`, an `except` clause handling it will catch any database access error.

As an alternative way of handling different error conditions, you can catch the `SPIError` exception and determine the specific error condition inside the `except` block by looking at the `sqlstate` attribute of the exception object. This attribute is a string value containing the "SQLSTATE" error code. This approach provides approximately the same functionality

43.8. Explicit Subtransactions

Recovering from errors caused by database access as described in Section 43.7.2 can lead to an undesirable situation where some operations succeed before one of them fails, and after recovering from that error the data is left in an inconsistent state. PL/Python offers a solution to this problem in the form of explicit subtransactions.

43.8.1. Subtransaction Context Managers

Consider a function that implements a transfer between two accounts:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'"
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

If the second `UPDATE` statement results in an exception being raised, this function will report the error, but the result of the first `UPDATE` will nevertheless be committed. In other words, the funds will be withdrawn from Joe's account, but will not be transferred to Mary's account.

To avoid such issues, you can wrap your `plpy.execute` calls in an explicit subtransaction. The `plpy` module provides a helper object to manage explicit subtransactions that gets created with the `plpy.subtransaction()` function. Objects created by this function implement the context manager interface. Using explicit subtransactions we can rewrite our function as:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'j
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'm
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Note that the use of `try/catch` is still required. Otherwise the exception would propagate to the top of the Python stack and would cause the whole function to abort with a PostgreSQL error, so that the `operations` table would not have any row inserted into it. The subtransaction context manager does not trap errors, it only assures that all database operations executed inside its scope will be atomically committed or rolled back. A rollback of the subtransaction block occurs on any kind of exception exit, not only ones caused by errors originating from database access. A regular Python exception raised inside an explicit subtransaction block would also cause the subtransaction to be rolled back.

43.8.2. Older Python Versions

Context managers syntax using the `with` keyword is available by default in Python 2.6. If using PL/Python with an older Python version, it is still possible to use explicit subtransactions, although not as transparently. You can call the subtransaction manager's `__enter__` and `__exit__` functions using the `enter` and `exit` convenience aliases. The example function that transfers funds could be written as:

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'j
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'm
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

> **Note:** Although context managers were implemented in Python 2.5, to use the `with` syntax in that version you need to use a future statement. Because of implementation details, however, you cannot use future statements in PL/Python functions.

43.9. Utility Functions

The `plpy` module also provides the functions `plpy.debug(``_msg_`), `plpy.log(``_msg_`), `plpy.info(``_msg_`), `plpy.notice(``_msg_`), `plpy.warning(``_msg_`), `plpy.error(``_msg_`), and `plpy.fatal(``_msg_`). `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. `raise plpy.Error(``_msg_`) and `raise plpy.Fatal(``_msg_`) are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other functions only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

Another set of utility functions are `plpy.quote_literal(``_string_`), `plpy.quote_nullable(``_string_`), and `plpy.quote_ident(``_string_`). They are equivalent to the built-in quoting functions described in Section 9.4. They are useful when constructing ad-hoc queries. A PL/Python equivalent of dynamic SQL from Example 40-1 would be:

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

43.10. Environment Variables

Some of the environment variables that are accepted by the Python interpreter can also be used to affect PL/Python behavior. They would need to be set in the environment of the main PostgreSQL server process, for example in a start script. The available environment variables depend on the version of Python; see the Python documentation for details. At the time of this writing, the following environment variables have an affect on PL/Python, assuming an adequate Python version:

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`
- `PYTHONVERBOSE`
- `PYTHONCASEOK`
- `PYTHONDONTWRITEBYTECODE`
- `PYTHONIOENCODING`
- `PYTHONUSERBASE`
- `PYTHONHASHSEED`

(It appears to be a Python implementation detail beyond the control of PL/Python that some of the environment variables listed on the `python` man page are only effective in a command-line interpreter and not an embedded Python interpreter.)

Chapter 44. 服务器编程接口

Table of Contents

服务器编程接口(SPI) 给在用户定义的C函数里面运行SQL 查询的能力。SQL是一套接口函数，用于简化对分析器、 规划器和执行器的访问。SQL还进行一些内存管理的工作。

> **Note:** 过程语言的存在也提供了其它的一些在过程里执行 SQL 命令的方法。 这些语言中的大部分本身就是基于 SPI 的， 因此这份文档可能会对那些语言的用户同样有帮助。

为了避免混淆，将使用术语"函数"(function)来代表 SPI接口函数，用"过程"(procedure) 代表用户用SPI定义的 C 函数。

注意，如果一条通过 SPI 调用的命令失败，那么控制不会返回到你的过程中。 取而代之的是，你的过程执行所在的事务或者子事务全部回滚。 这一点看起来可能很奇怪，因为大多数 SPI 函数的文档里都有错误返回习惯。 不过，那些习惯只适用于在 SPI 函数自己内部检测到的错误。 可以通过在你自己的可能失败的 SPI 调用周围建立一个子事务的方法来在错误之后恢复控制。 目前还没有写这方面的文档，因为所需要的机制仍然在变化。

如果执行成功了，SPI函数返回一个非负结果 (或者通过返回一个整数值或放在全局变量 `SPI_result` 里， 像下面描述的那样)。出错时，返回一个负数或 `NULL` 结果。

使用 SPI 的源代码文件必须包含头文件 `executor/spi.h` 。

44.1. 接口函数

Table of Contents

SPI_connect

Name

SPI_connect -- 把一个过程与 SPI 管理器连接起来

Synopsis

```
int SPI_connect(void)
```

描述

`SPI_connect` 打开一个从过程调用到 SPI 管理器的连接。 如果你需要通过 SPI 执行命令，你就必需调用这个函数。 有些工具类 SPI 函数可以从未连接的过程中调用。

如果你的过程已经连接了，那么 `SPI_connect` 将返回一个 SPI_ERROR_CONNECT错误信息。 请注意如果一个过程已经调用了 `SPI_connect` 然后它直接调用另外一个又会调用 `SPI_connect` 的过程的时候也会发生这种问题。尽管以一个 SQL 命令里调用另外一个使用SPI的函数的形式对SPI 管理器进行递归调用是允许的，但是直接的嵌套调用 `SPI_connect` 和 `SPI_finish` 是不允许的(不过，可以看看 `SPI_push` 和 `SPI_pop`)。

返回值

`SPI_OK_CONNECT`

成功时

`SPI_ERROR_CONNECT`

失败时

SPI_finish

Name

SPI_finish -- 把一个过程与 SPI 管理器断开

Synopsis

```
int SPI_finish(void)
```

描述

`SPI_finish` 关闭一个现有的到 SPI 管理器的连接。 在完成你的过程的当前调用所必须的 SPI 操作之后，你必须调用这个函数。 不过，如果你通过 `elog(ERROR)` 退出事务， 那么你就不需要担心这件事情。在这种情况下，SPI 将自动清理干净。

如果 `SPI_finish` 是在当前没有有效连接的情况下被调用的， 你可能会得到一个 `SPI_ERROR_UNCONNECTED` 的返回。 这样做没有什么根本性问题，这意味着 SPI 管理器不做任何事情。

返回值

`SPI_OK_FINISH`

如果正常断开

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的过程调用

SPI_push

Name

SPI_push -- 对 SPI 堆栈进行压栈操作以允许递归的 SPI 使用

Synopsis

```
void SPI_push(void)
```

描述

在执行另外一个可能也使用了 SPI 的过程之前，应该调用 `SPI_push`。 之后，SPI 不再是"已连接"状态，除非再次进行 `SPI_push`， 否则 SPI 函数调用将被拒绝。这样就保证了你的过程的 SPI 状态和另外一个你调用的过程的状态之间的干净的隔离。在另外一个过程返回后， 调用 `SPI_pop` 恢复对你自己的 SPI 状态的访问。

请注意 `SPI_execute` 和相关的函数在把控制交回 SQL 执行引擎之前自动做与 `SPI_push` 相当的工作， 因此你在使用这些函数的时候不用担心这些。只有在你直接调用任意可能包含 `SPI_connect` 调用的代码的时候，你才需要发出 `SPI_push` 和 `SPI_pop` 。

SPI_pop

Name

SPI_pop -- 弹出 SPI 堆栈以从递归的 SPI 使用中返回

Synopsis

```
void SPI_pop(void)
```

描述

`SPI_pop`从 SPI 调用堆栈里弹出前面的环境。 参阅 `SPI_push` 。

SPI_execute

Name

SPI_execute -- 执行一条命令

Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

描述

`SPI_execute` 执行声明的 SQL 命令获取 `count` 行。如果 `read_only` 为 `true`，命令必须是只读的， 因此可以略微降低一些执行的开销。

这个函数只能在已连接的过程中调用。

如果 `count` 是零，则在命令适合的所有行上执行。 如果 `count` 大于 0， 那么将不会超过 `count` 行被检索；当达到计数时执行停止， 很像在查询中添加了一个 `LIMIT` 子句。比如，

```
SPI_execute("SELECT * FROM foo", true, 5);
```

将从表中最多检索5行。请注意，这样一个限制只在命令实际返回行时有效。例如，

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

插入 `bar` 中的所有行，忽略 `count` 参数。 不过，

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

将最多插入5行，因为在检索到第5个 `RETURNING` 结果行之后执行将停止。

你可以在一个字符串里传递多个命令。 `SPI_execute` 返回最后执行的命令的结果。 `count` 的限制独立地应用于每一个命令（即使实际只返回最后的结果）。 限制不会应用于规则生成的隐藏命令。

如果 `read_only` 是 `false`， `SPI_execute` 递增命令计数器并且在字符串里执行每个命令之前计算一个新的快照。 如果当前事务的隔离级别是 `SERIALIZABLE` 或 `REPEATABLE READ`， 这个快照实际上并不改变，但是在 `READ COMMITTED` 模式里， 这个快照更新允许每个命令看到其它会

话的新提交的事务的结果。 这样实际上是为了修改数据库的命令有一致的行为。

如果 `read_only` 是 `true`， `SPI_execute` 并不更新快照或者命令计数器，并且它只允许简单的 `SELECT` 命令出现在命令字符串里。这个命令使用为周围的查询建立起来的快照执行。 这个执行模式比读/写模式执行得略微块些，因为它消除了每个命令的一些开销。 并且它还允许制作真正的稳定函数：因为随后的执行都将使用同一个快照， 结果里不会有改变。

通常，在同一个使用 SPI 的函数里混杂只读和读写命令是不明智的；那样可能导致非常混乱的行为，因为只读的查询不能看到任何读写的查询做的数据库更新。

(最后)一条命令执行返回的结果的实际行数会放在全局的变量 `SPI_processed` 里。如果函数的返回值是 `SPI_OK_SELECT` 、 `SPI_OK_INSERT_RETURNING` 、 `SPI_OK_DELETE_RETURNING` 或 `SPI_OK_UPDATE_RETURNING` ，那么你可以使用全局指针 `SPITupleTable *SPI_tuptable` 访问结果行。一些实用命令 (比如 `EXPLAIN`)还返回行集合，并且 `SPI_tuptable` 也将在这种情况下包含结果。一些实用命令（ `COPY` ， `CREATE TABLE AS` ） 并不返回行集，所以 `SPI_tuptable` 为 NULL，但是它们仍然返回 `SPI_processed` 中处理了的行数。

结构 `SPITupleTable` 是这样定义的：

```
typedef struct
{
    MemoryContext tuptabcxt;    /* memory context of result table */
    uint32      alloced;        /* number of alloced vals */
    uint32      free;           /* number of free vals */
    TupleDesc   tupdesc;        /* row descriptor */
    HeapTuple  *vals;           /* rows */
} SPITupleTable;
```

`vals` 是一个指向数据行的的指针数组(有效记录的数目由 `SPI_processed` 给出)。 `tupdesc` 是一个行描述符， 你可以传递给 SPI 函数处理这些数据行。 `tuptabcxt` 、 `alloced` 和 `free` 是 SPI 的内部字段， 并非给 SPI 调用者使用的。

`SPI_finish` 释放所有在当前过程中分配的 `SPITupleTable` 。 如果你已经处理完特定的结果表，那么可以更早地释放它，方法是调用 `SPI_freetuptable` 。

参数

`const char *` `command`

包含要执行的命令的字符串

`bool` `read_only`

`true` 用于只读的执行

`long` `count`

返回的最大行数，或者没有限制时为 `0`

返回值

如果命令执行成功，那么返回下列值之一(非负数)：

`SPI_OK_SELECT`

如果执行了一个 `SELECT` 但不是 `SELECT INTO`

`SPI_OK_SELINTO`

如果执行了一条 `SELECT INTO`

`SPI_OK_INSERT`

如果执行了一条 `INSERT`

`SPI_OK_DELETE`

如果执行了一条 `DELETE`

`SPI_OK_UPDATE`

如果执行了一条 `UPDATE`

`SPI_OK_INSERT_RETURNING`

如果执行了一条 `INSERT RETURNING`

`SPI_OK_DELETE_RETURNING`

如果执行了一条 `DELETE RETURNING`

`SPI_OK_UPDATE_RETURNING`

如果执行了一条 `UPDATE RETURNING`

`SPI_OK_UTILITY`

如果执行了一条实用命令(比如 `CREATE TABLE`)

`SPI_OK_REWRITTEN`

如果该命令通过一个规则重新写入了另一个类型的命令 （比如， `UPDATE` 变成一个 `INSERT` ）

发生错误时，返回下列负数值之一：

`SPI_ERROR_ARGUMENT`

如果 `command` 是 `NULL` 或 `count` 小于 0

`SPI_ERROR_COPY`

如果企图进行 `COPY TO stdout` 或 `COPY FROM stdin`

`SPI_ERROR_TRANSACTION`

如果尝试事务操纵命令(`BEGIN` , `COMMIT` , `ROLLBACK` , `SAVEPOINT` , `PREPARE TRANSACTION` , `COMMIT PREPARED` , `ROLLBACK PREPARED` , 或它们的变种)

`SPI_ERROR_OPUNKNOWN`

命令类型未知(不应该发生)

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的过程中调用

注意

所有SPI查询执行函数设置了 `SPI_processed` 和 `SPI_tuptable` (只是一个指针，不是结构的内容)。如果你需要跨越后面的调用访问 `SPI_execute` 或者另一个查询执行函数的结果表，那么需要把这两个全局变量保存到一个局部过程变量中。

SPI_exec

Name

SPI_exec -- 执行一个读/写命令

Synopsis

```
int SPI_exec(const char * command, long count)
```

描述

`SPI_exec` 和 `SPI_execute` 一样，只是相当于后者的 `read_only` 参数总是 `false` 。

参数

`const char *` `command`

包含需要执行的命令的字符串

`long` `count`

返回的最大行数，或者没有限制时为 `0`

返回值

参见 `SPI_execute` 。

SPI_execute_with_args

Name

SPI_execute_with_args -- 执行一个带有外联参数的命令

Synopsis

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

描述

`SPI_execute_with_args` 执行一个可能包含引用外部提供的参数的命令。 该命令文本作为 `$``_n_` 引用一个参数， 并且该调用为每个这样的符号指定数据类型和值。 `read_only` 和 `count` 的解释和在 `SPI_execute` 中一样。

与 `SPI_execute` 比较，这个例程的主要优势是数据值可以插入命令， 而不用引用/逃逸，并且因此少了许多SQL注入攻击的危险。

相似的结果可以用跟着 `SPI_execute_plan` 的 `SPI_prepare` 达到；不过，当使用这个函数时，该查询规划总是自定义为提供的特定的参数值。 对于一次性查询执行，这个函数应该优先执行。如果用许多不同的参数执行相同的命令， 哪种方法可能会更快，取决于重新规划的开销与自定义规划的好处的对比。

参数

`const char *` `command`

命令字符串

`int` `nargs`

输入参数的个数 (`$1` , `$2` , 等)

`Oid *` `argtypes`

长度 `nargs` 的一个数组，包含参数的数据类型的OID

`Datum *` `values`

长度 `nargs` 的一个数组，包含实际参数值

`const char *` `nulls`

长度 `nargs` 的一个数组，描述哪个参数为空

如果 `nulls` 是 `NULL` ，那么 `SPI_execute_with_args` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '`，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。（在后面这种情况下，对应的 `values` 项中的实际值无关紧要。） 请注意， `nulls` 不是文本字符串，只是一个数组： 它不需要 `'\0'` 终止符。

`bool` `read_only`

`true` 用于只读的执行

`long` `count`

返回的最大行数，或者没有限制时为 `0`

返回值

返回值和 `SPI_execute` 相同。

如果成功， `SPI_processed` 和 `SPI_tuptable` 和在 `SPI_execute` 中一样设置。

SPI_prepare

Name

SPI_prepare -- 准备一个规划但不立即执行它

Synopsis

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

描述

`SPI_prepare` 为声明的命令创建和返回一个预备语句但是不执行查询。 该预备语句稍后可以使用 `SPI_execute_plan` 重复的执行。

如果相同或者类似的查询要多次重复执行，那么通常只进行一次解析分析应该是更好些， 并且此外可能有利于为该命令重复使用一个执行规划。 `SPI_prepare` 把一个命令字符串转换成一个封装解析分析的结果的预备语句。 如果发现为每个执行生成一个自定义规划没什么帮助， 那么该预备语句也为缓存一个执行规划提供位置。

可以把预编写的查询通用化，方法是在那些普通查询里是常量的地方书写参数 (`$1` , `$2` 等等)。参数的数值随后在调用 `SPI_execute_plan` 的时候声明。这样就允许已准备的查询在远比没有参数时广泛得多的场合下使用。

`SPI_prepare` 返回的语句只能在当前过程调用中使用， 因为 `SPI_finish` 释放为这样一个语句分配的内存。不过， 一个语句可以用函数 `SPI_keepplan` 或 `SPI_saveplan` 保存更长的时间。

参数

`const char *` `command`

命令字符串

`int` `nargs`

输入参数的个数(`$1` , `$2` 等等)

`Oid *` `argtypes`

一个指针，指向包含参数数据类型的OID数组

返回值

`SPI_prepare` 返回一个指向一个 `SPIPlan` 的非空指针， `SPIPlan` 是一个表示预备语句的不透明结构。错误时将返回 `NULL` ， 并且 `SPI_result` 将设置为和 `SPI_execute` 使用的同样错误的错误代码，例外是在 `command` 是 `NULL` 的时候，或者是 `nargs` 小于 0 或者 `nargs` 大于 0 并且 `argtypes` 是 `NULL` 的时候会被设置成 `SPI_ERROR_ARGUMENT` 。

注意

如果没有定义参数，那么在第一次使用 `SPI_execute_plan` 时将创建一个通用规划，并且也用于所有随后的执行。如果有参数， 前几次使用 `SPI_execute_plan` 将生成特定于提供的参数值的自定义规划。 相同的预备语句使用足够多次之后， `SPI_execute_plan` 将建立一个通用规划，并且如果该通用规划并不比自定义规划昂贵的多的话， 它将开始使用该通用规划而不是每次重新规划。如果这个缺省行为不合适， 你可以通过传递 `CURSOR_OPT_GENERIC_PLAN` 或 `CURSOR_OPT_CUSTOM_PLAN` 标志到 `SPI_prepare_cursor` 来修改它，分别强制使用通用或自定义规划。

尽管预备语句的要点是为了避免重复的解析分析和规划语句， 但是在语句中使用的数据库对象自上次使用预备语句以来经历了明确的（DDL） 改变时，PostgreSQL将在使用它之前强制重新分析和重新规划语句。 另外，如果search_path的值在下一次使用时发生了改变时， 该语句将使用新的 `search_path` 重新解析。 （后者的行为是 PostgreSQL 9.3新增的。）参阅PREPARE 获取更多关于预备语句的行为的信息。

这个函数应该只从一个已连接的过程中调用。

`SPIPlanPtr` 声明为一个指针，指向 `spi.h` 中的一个不透明的结构类型。 尝试直接访问它的内容是不明智的，因为那样会使得你的代码在未来的 PostgreSQL修订版本中更容易破裂。

名字 `SPIPlanPtr` 是历史用法，因为数据结构不再需要包含一个执行规划。

SPI_prepare_cursor

Name

SPI_prepare_cursor -- 准备一个语句但不立即执行它

Synopsis

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

描述

`SPI_prepare_cursor` 和 `SPI_prepare` 相同， 除了它也允许说明规划器的"游标选项"参数。这是一个位掩码， 让 `DeclareCursorStmt` 的 `options` 字段拥有 `nodes/parsenodes.h` 中显示的值。 `SPI_prepare` 总是将游标选项看做零。

参数

`const char *` `command`

命令字符串

`int` `nargs`

输入参数的个数(`$1` , `$2` 等等)

`Oid *` `argtypes`

一个指针，指向一个包含参数数据类型的OID的数组

`int` `cursorOptions`

游标选项的整数位标记；零表示缺省行为

返回值

`SPI_prepare_cursor` 有和 `SPI_prepare` 一样的返回约定。

注意

`cursorOptions` 中的有效位包括 `CURSOR_OPT_SCROLL` 、 `CURSOR_OPT_NO_SCROLL` 、 `CURSOR_OPT_FAST_PLAN` 、 `CURSOR_OPT_GENERIC_PLAN` 和 `CURSOR_OPT_CUSTOM_PLAN` 。 特别要注意忽略了 `CURSOR_OPT_HOLD` 。

SPI_prepare_params

Name

SPI_prepare_params -- 准备一个语句但不立即执行它

Synopsis

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

描述

`SPI_prepare_params` 为指定的命令创建并返回一个预备语句，但是不执行该命令。这个函数相当于 `SPI_prepare_cursor`，除了调用者可以指定解析器hook函数来控制外部参数引用的解析。

参数

`const char *` `command`

命令字符串

`ParserSetupHook` `parserSetup`

解析器hook设置功能

`void *` `parserSetupArg`

为 `parserSetup` 传递参数

`int` `cursorOptions`

游标选项的整型位标记；零表示缺省行为

返回值

`SPI_prepare_params` 有和 `SPI_prepare` 一样的返回约定。

SPI_getargcount

Name

SPI_getargcount -- 返回一个 `SPI_prepare` 准备的已准备好语句需要的参数个数

Synopsis

```
int SPI_getargcount(SPIPlanPtr plan)
```

描述

`SPI_getargcount` 返回执行一个 `SPI_prepare` 准备的已准备好语句需要的参数个数

参数

`SPIPlanPtr` `plan`

预备语句（ `SPI_prepare` 返回的）

返回值

`plan` 预期参数的个数。如果 `plan` 是 `NULL` 或无效， `SPI_result` 设置为 `SPI_ERROR_ARGUMENT` 并返回-1.

SPI_getargtypeid

Name

SPI_getargtypeid -- 返回 `SPI_prepare` 准备的已准备好语句的参数的数据类型OID

Synopsis

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

描述

`SPI_getargtypeid` 返回 `SPI_prepare` 准备的已准备好语句的第 `argIndex` 个参数的类型 ID，用 OID 表示。第一个参数的索引为零。

参数

`SPIPlanPtr` `plan`

预备语句（`SPI_prepare` 返回的）

`int` `argIndex`

以零为基的参数索引

返回值

给出索引位置的参数的类型OID，如果 `plan` 为 `NULL` 或无效，或 `argIndex` 小于 0 或者大于或等于为 `plan` 声明的参数个数，则 `SPI_result` 设置为 `SPI_ERROR_ARGUMENT` 并且返回 `InvalidOid` 。

SPI_is_cursor_plan

Name

SPI_is_cursor_plan -- 如果一个 `SPI_prepare` 准备的语句可以和 `SPI_cursor_open` 一起使用，则返回 `true`

Synopsis

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

描述

如果一个 `SPI_prepare` 准备的语句可以作为参数传递给 `SPI_cursor_open` 则 `SPI_is_cursor_plan` 返回 `true` ，如果不是这样则返回 `false` 。 评判的标准是这个 `plan` 代表一个单个命令， 并且这个命令返回元组；例如一个包含 `INTO` 子句的 `SELECT` 或者包含 `RETURNING` 子句的 `UPDATE` 。

参数

`SPIPlanPtr` `plan`

预备语句（ `SPI_prepare` 返回的）

返回值

表明该 `plan` 是否可以生成一个游标的 `true` 或 `false` ， `SPI_result` 设置为零。 如果不可能确定结果（例如，如果 `plan` 是 `NULL` 或无效，或者如果在没有连接到SPI时调用）， 那么 `SPI_result` 设置为合适的错误代码，并且返回 `false` 。

SPI_execute_plan

Name

SPI_execute_plan -- 执行一个 `SPI_prepare` 准备的语句

Synopsis

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

描述

`SPI_execute_plan` 执行一个 `SPI_prepare` 或它的兄弟节点之一准备的语句。 `read_only` 和 `count` 的含义和 `SPI_execute` 里面的相同。

参数

`SPIPlanPtr` `plan`

预备语句（ `SPI_prepare` 返回的）

`Datum *` `values`

一个实际的参数值的数组。必须和语句的参数个数相同。

`const char *` `nulls`

一个描述哪个参数是空的数组。必须和语句的参数个数相同。

如果 `nulls` 是 `NULL` ，那么 `SPI_execute_plan` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '`，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。 （在后面这种情况下，对应的 `values` 项中的实际值无关紧要。） 请注意， `nulls` 不是文本字符串，只是一个数组： 它不需要 `'\0'` 终止符。

`bool` `read_only`

`true` 用于只读的执行

`long` `count`

返回的最大行数，或者没有限制时为 `0`

返回值

返回值和 `SPI_execute` 的一样，另外还有下面几个可能的错误(负值)结果：

`SPI_ERROR_ARGUMENT`

如果 `plan` 是 `NULL` 或者无效， 或者 `count` 小于 0

`SPI_ERROR_PARAM`

如果 `values` 是 `NULL` 并且 `plan` 准备了一些参数

成功时， `SPI_processed` 和 `SPI_tuptable` 的设置和 `SPI_execute` 里一样。

SPI_execute_plan_with_paramlist

Name

SPI_execute_plan_with_paramlist -- 执行一个`SPI_prepare`准备的已准备好的语句

Synopsis

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

描述

`SPI_execute_plan_with_paramlist`执行一个`SPI_prepare`准备的已准备好的语句。这个函数相当于`SPI_execute_plan`，除了要传递给查询的有关参数值信息的表现不同。`ParamListInfo`表示法可以方便的传递早已在该格式中可用的值。它也支持使用动态参数设置，通过`ParamListInfo`中指定的hook函数。

参数

`SPIPlanPtr` `plan`

预备语句（`SPI_prepare`返回的）

`ParamListInfo` `params`

包含参数类型和值的数据结构；如果没有则为NULL

`bool` `read_only`

`true`用于只读的执行

`long` `count`

返回的最大行数，或者没有限制时为`0`

返回值

返回值和 `SPI_execute_plan` 的相同。

如果成功， `SPI_processed` 和 `SPI_tuptable` 的设置和 `SPI_execute_plan` 里的相同。

SPI_execp

Name

SPI_execp -- 以读/写模式执行一个准备的查询规划

Synopsis

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

描述

`SPI_execp` 和 `SPI_execute_plan` 一样， 只是后者的 `read_only` 参数总是为 `false` 。

参数

`SPIPlanPtr` `plan`

预备语句（ `SPI_prepare` 返回的）

`Datum *` `values`

实际的参数值的数组，必须和语句的参数个数一样。

`const char *` `nulls`

一个描述哪个参数是空的数组。必须和参数的个数一样。

如果 `nulls` 是 `NULL` ，那么 `SPI_execp` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '` ，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。 （在后面这种情况下，对应的 `values` 项中的实际值无关紧要。） 请注意， `nulls` 不是文本字符串，只是一个数组： 它不需要 `'\0'` 终止符。

`long` `count`

返回的最大行数，或者没有限制时为 `0`

返回值

参阅 `SPI_execute_plan` 。

成功时， `SPI_processed` 和 `SPI_tuptable` 的设置和 `SPI_execute` 里一样。

SPI_cursor_open

Name

SPI_cursor_open -- 用 `SPI_prepare` 创建的语句设置一个游标

Synopsis

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

描述

`SPI_cursor_open` 设置一个游标(内部叫入口)， 这个游标可以执行 `SPI_prepare` 准备的语句。参数和对应的 `SPI_execute_plan` 参数具有相同的含义。

使用游标而不是直接执行语句有两个优点。首先，结果行可以每次检索一小部分， 避免那些返回大量数据行的查询造成的内存缺乏。第二， 一个入口可以在当前过程之外存活(实际上，它可以活到当前事务的结尾)。 给过程的调用者返回一个入口名是一种返回行结果集的方法。

传入参数数据将被拷贝到游标的入口，所以它在游标仍然存在时可以被释放。

参数

`const char *` `name`

入口的名字，或者是用 `NULL` 让系统选择一个名字

`SPIPlanPtr` `plan`

预备语句（`SPI_prepare` 返回的）

`Datum *` `values`

一个实际参数值的数组。必须和语句的参数个数相等。

`const char *` `nulls`

一个描述哪些参数是 NULL 的数组。必须和语句的参数个数相等。

如果 `nulls` 是 `NULL` ，那么 `SPI_cursor_open` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '`，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。（在后面这种情况下，对应的 `values` 项中的实际值无关紧要。）请注意， `nulls` 不是文本字符串，只是一个数组：它不需要 `'\0'` 终止符。

`bool` `read_only`

`true` 用于只读的执行

返回值

指向包含游标入口的指针。请注意，这里没有错误返回约定；任何错误都将通过 `elog` 报告。

SPI_cursor_open_with_args

Name

SPI_cursor_open_with_args -- 使用查询和参数设置一个游标

Synopsis

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

描述

`SPI_cursor_open_with_args` 设置一个游标（内部的，一个入口）， 该游标将执行指定的查询。大多数参数和 `SPI_prepare_cursor` 和 `SPI_cursor_open` 的对应参数有相同的含义。

对于一次性查询执行，跟着 `SPI_cursor_open` 的 `SPI_prepare_cursor` 应该会偏爱这个函数。 如果相同的命令用不同的参数执行，哪种方法可能更快， 取决于重新规划的开销和自定义规划的益处的比较。

传入参数数据将被拷贝到游标的入口，所以它在游标仍然存在时可以被释放。

参数

`const char *` `name`

入口的名字，或 `NULL` 让系统选择一个名字

`const char *` `command`

命令字符串

`int` `nargs`

输入参数的个数 (`$1` , `$2` 等等)

`Oid *` `argtypes`

长度 `nargs` 的一个数组，包括参数数据类型的OID

`Datum *` `values`

长度 `nargs` 的一个数组，包括实际的参数值

`const char *` `nulls`

长度 `nargs` 的一个数组，描述哪个参数为空

如果 `nulls` 是 `NULL` ，那么 `SPI_cursor_open_with_args` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '`，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。（在后面这种情况下，对应的 `values` 项中的实际值无关紧要。） 请注意， `nulls` 不是文本字符串，只是一个数组： 它不需要 `'\0'` 终止符。

`bool` `read_only`

`true` 用于只读的执行

`int` `cursorOptions`

游标选项的整型位标记；零表示缺省行为

返回值

指向包含游标入口的指针。请注意，这里没有错误返回约定； 任何错误都将通过 `elog` 报告。

SPI_cursor_open_with_paramlist

Name

SPI_cursor_open_with_paramlist -- 使用参数设置一个游标

Synopsis

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

描述

`SPI_cursor_open_with_paramlist` 设置一个游标（内部的，一个入口）， 该游标将执行一个 `SPI_prepare` 准备的已准备好的语句。 这个函数相当于 `SPI_cursor_open` ， 除了要传递给查询的有关参数值信息的表现不同。 `ParamListInfo` 表示法可以方便的传递早已在该格式中可用的值。它也支持使用动态参数设置， 通过 `ParamListInfo` 中指定的hook函数。

传入参数数据将被拷贝到游标的入口，所以它在游标仍然存在时可以被释放。

参数

`const char *` `name`

入口的名字，或者是 `NULL` 让系统选择一个名字

`SPIPlanPtr` `plan`

预备语句（ `SPI_prepare` 返回的）

`ParamListInfo` `params`

包含参数类型和值的数据结构；如果没有则为NULL

`bool` `read_only`

`true` 用于只读的执行

返回值

指向包含游标入口的指针。请注意，这里没有错误返回约定；任何错误都将通过 `elog` 报告。

SPI_cursor_find

Name

SPI_cursor_find -- 用名字寻找并执行一个现存的游标

Synopsis

```
Portal SPI_cursor_find(const char * name)
```

描述

`SPI_cursor_find`通过名字寻找一个现存的入口。 主要用于解析一些其它函数返回的文本的游标名字。

参数

`const char *` `name`

入口的名字

返回值

指向指定名称的入口的指针，如果没有找到就是`NULL`

SPI_cursor_fetch

Name

SPI_cursor_fetch -- 从一个游标里抓取一些行

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

描述

`SPI_cursor_fetch` 从一个游标里抓取一些行。 这个函数等效于 SQL 命令 `FETCH` 的一个子集（参阅 `SPI_scroll_cursor_fetch` 获取更多功能性）。

参数

`Portal` `portal`

包含游标的入口

`bool` `forward`

向前抓取时为真，向后抓取时为假

`long` `count`

要抓取的最大行数

返回值

成功时，像 `SPI_execute` 那样设置 `SPI_processed` 和 `SPI_tuptable`

注意

如果游标的规划没有带有 `CURSOR_OPT_SCROLL` 选项创建， 那么向后抓取会失败。

SPI_cursor_move

Name

SPI_cursor_move -- 移动一个游标

Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

描述

`SPI_cursor_move`忽略游标中的一些行。 这个函数等效于 SQL 命令`MOVE`的一个子集（参阅`SPI_scroll_cursor_move`获取更多功能性）。

参数

`Portal portal`

包含游标的入口

`bool forward`

向前移动时为真，向后移动时为假

`long count`

移动的最大行数

注意

如果游标的规划没有带有`CURSOR_OPT_SCROLL`选项创建， 那么向后移动会失败。

SPI_scroll_cursor_fetch

Name

SPI_scroll_cursor_fetch -- 从一个游标中抓取一些行

Synopsis

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

描述

`SPI_scroll_cursor_fetch` 从一个游标中抓取一些行。 这个函数等效于SQL命令的 `FETCH` 。

参数

`Portal` `portal`

包含游标的入口

`FetchDirection` `direction`

`FETCH_FORWARD` 、 `FETCH_BACKWARD` 、 `FETCH_ABSOLUTE` 或 `FETCH_RELATIVE` 之一

`long` `count`

`FETCH_FORWARD` 或 `FETCH_BACKWARD` 抓取的最大行数； `FETCH_ABSOLUTE` 抓取的绝对行数；或 `FETCH_RELATIVE` 抓取的相对行数。

返回值

如果成功， `SPI_processed` 和 `SPI_tuptable` 的设置和 `SPI_execute` 里一样。

注意

参阅SQLFETCH命令获取 `direction` 和 `count` 参数的解释的详细信息。

如果游标的规划没有带有`CURSOR_OPT_SCROLL`选项创建，那么方向值而不是`FETCH_FORWARD`会失败。

SPI_scroll_cursor_move

Name

SPI_scroll_cursor_move -- 移动一个游标

Synopsis

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

描述

`SPI_scroll_cursor_move` 忽略游标中的一些行数。 这个函数相当于SQL命令 `MOVE` 。

参数

`Portal` `portal`

包含游标的入口

`FetchDirection` `direction`

`FETCH_FORWARD` 、 `FETCH_BACKWARD` 、 `FETCH_ABSOLUTE` 或 `FETCH_RELATIVE` 之一

`long` `count`

`FETCH_FORWARD` 或 `FETCH_BACKWARD` 移动的行数； `FETCH_ABSOLUTE` 移动的绝对行数；或 `FETCH_RELATIVE` 移动的相对行数

返回值

如果成功， `SPI_processed` 的设置和 `SPI_execute` 里的一样。 `SPI_tuptable` 设置为 `NULL` ，因为这个函数没有返回行。

注意

参阅SQL FETCH命令获取 `direction` 和 `count` 参数的解释的详细信息。

如果游标的规划没有使用`CURSOR_OPT_SCROLL`选项创建， 那么方向值而不是`FETCH_FORWARD`会失败。

SPI_cursor_close

Name

SPI_cursor_close -- 关闭一个游标

Synopsis

```
void SPI_cursor_close(Portal portal)
```

描述

`SPI_cursor_close` 关闭一个前面创建的游标并且释放其入口存储。

在事务结尾，所有打开的游标都自动关闭。只有在希望更早释放资源的时候才需要调用 `SPI_cursor_close` 。

参数

`Portal` `portal`

包含游标的入口

SPI_keepplan

Name

SPI_keepplan -- 保存一个预备语句

Synopsis

```
int SPI_keepplan(SPIPlanPtr plan)
```

描述

`SPI_keepplan` 保存一个已经传递了的语句（ `SPI_prepare` 准备的），所以它将不会通过 `SPI_finish` 或者事务管理器释放。 这给了你在当前会话中你的过程的随后调用重新使用预备语句的能力。

参数

`SPIPlanPtr` `plan`

要保存的预备语句

返回值

成功时为0；如果 `plan` 是 `NULL` 或者无效的， 那么是 `SPI_ERROR_ARGUMENT`

注意

通过指针调整，传入参数重定位到参数存储（不需要拷贝数据）。 如果你稍后想要删除它，在其上使用 `SPI_freeplan` 。

SPI_saveplan

Name

SPI_saveplan -- 保存一个预备语句

Synopsis

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

描述

`SPI_saveplan` 在内存里保存一个传递进来的语句(用 `SPI_prepare` 准备的)，它将不会被 `SPI_finish` 和事务管理器释放，并且返回一个指向拷贝的语句的指针。 这样就给你在当前会话里的随后的调用中复用这个预备规划的能力。

参数

`SPIPlanPtr` `plan`

要保存的预备语句

返回值

指向保存的规划的指针；如果不成功则为 NULL 。出错的时候，像下面这样设置 SPI_result ：

`SPI_ERROR_ARGUMENT`

如果 `plan` 是 `NULL` 或者无效的

`SPI_ERROR_UNCONNECTED`

如果从一个未连接的过程中调用

注意

原先传入的语句并没有释放，所以你可能希望在其上执行 `SPI_freeplan` ， 以避免内存泄露直到 `SPI_finish` 。

在大多数情况下， `SPI_keepplan` 偏爱这个函数， 因为它基本上不需要物理上拷贝预备语句的数据结构就完成了相同的结果。

44.2. 接口支持函数

Table of Contents

这里描述的函数提供了从 `SPI_execute` 返回的结果集以及其它 SPI 接口函数中抽取信息的便利的接口。

所有本节描述的函数都可以用于已连接和未连接的过程。

SPI_fname

Name

SPI_fname -- 从指定的字段编号判断字段名字

Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_fname` 返回指定字段的字段名字。你可以在不在需要的时候用 `pfree` 释放这个名字的副本占据的空间。

参数

`TupleDesc` `rowdesc`

输入行描述

`int` `colnumber`

字段编号(从 1 开始)

返回值

字段名；如果 `colnumber` 超出范围则返回 `NULL` 。 错误时， `SPI_result` 设置为 `SPI_ERROR_NOATTRIBUTE` 。

SPI_fnumber

Name

SPI_fnumber -- 判断声明字段名的字段编号

Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

描述

`SPI_fnumber` 返回声明名字的字段的字段编号。

如果 `colname` 引用的是一个系统字段(比如 `oid`)， 那么将返回合适的负数字段编号。调用者应该仔细测试返回值是 `SPI_ERROR_NOATTRIBUTE` 才能判断是一个错误；除非要拒绝系统字段，否则，测试结果小于或者等于 0 是不正确的判断方法。

参数

`TupleDesc` `rowdesc`

输入行描述

`const char *` `colname`

字段名

返回值

字段编号(从 1 开始记)，如果没有找到该名字的字段， 返回 `SPI_ERROR_NOATTRIBUTE`

SPI_getvalue

Name

SPI_getvalue -- 返回声明字段的字符串值

Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

描述

`SPI_getvalue` 返回声明的字段的数值的字符串表现形式。

结果是放在用 `palloc` 分配的内存里的。 在你不需要它之后，你可以使用 `pfree` 释放内存。

参数

`HeapTuple` `row`

要检查的输入行

`TupleDesc` `rowdesc`

输入行描述

`int` `colnumber`

字段编号(从 1 开始)

返回值

字段值，如果字段是空，或者 `colnumber` 超出范围 (`SPI_result` 设置为 `SPI_ERROR_NOATTRIBUTE`)， 或者没有可用的输出函数(`SPI_result` 设置为 `SPI_ERROR_NOOUTFUNC`)的时候，返回 `NULL` 。

SPI_getbinval

Name

SPI_getbinval -- 返回声明字段的二进制值

Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

描述

`SPI_getbinval` 以数据的内部形式(像类型 `Datum`) 返回声明字段的数值。

这个函数并不为数据分配新的空间。如果是传递引用，那么返回值将是指向传递的行的指针。

参数

`HeapTuple` `row`

输入的要检查的行

`TupleDesc` `rowdesc`

输入行描述

`int` `colnumber`

字段编号(从 1 开始记)

`bool *` `isnull`

字段里 NULL 的标志

返回值

返回该字段的二进制值。如果字段为空，那么 `isnull` 指向的变量设置为真，否则为假。

错误时， `SPI_result` 设置为 `SPI_ERROR_NOATTRIBUTE` 。

SPI_gettype

Name

SPI_gettype -- 返回声明字段的数据类型名

Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_gettype` 返回声明字段的数据类型名字的一份拷贝。 如果你不再需要它了，你可以使用 `pfree` 释放名字的拷贝。

参数

`TupleDesc` `rowdesc`

输入行描述

`int` `colnumber`

字段编号(从 1 开始记)

返回值

声明的字段的数据类型名，如果错误，则为 `NULL` 。 错误时， `SPI_result` 设置为 `SPI_ERROR_NOATTRIBUTE` 。

SPI_gettypeid

Name

SPI_gettypeid -- 返回声明字段的数据类型OID

Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

描述

`SPI_gettypeid` 返回声明字段的数据类型的OID。

参数

`TupleDesc` `rowdesc`

输入行描述

`int` `colnumber`

字段编号(从 1 开始记)

返回值

声明字段的数据类型的OID，或者错误时是 `InvalidOid` 。 在出错的时候， `SPI_result` 设置为 `SPI_ERROR_NOATTRIBUTE` 。

SPI_getrelname

Name

SPI_getrelname -- 返回声明关系的名字

Synopsis

```
char * SPI_getrelname(Relation rel)
```

描述

`SPI_getrelname` 返回声明的关系的名字之一份拷贝。 如果你不再需要这个名字，可以用 `pfree` 释放这个拷贝。

参数

`Relation` `rel`

输入关系

返回值

声明的关系的名字。

SPI_getnspname

Name

SPI_getnspname -- 返回声明关系的名字空间

Synopsis

```
char * SPI_getnspname(Relation rel)
```

描述

`SPI_getnspname` 返回一份声明的 `Relation` 所属的名字空间的名字的拷贝。这个数值等于该关系的模式。在你使用完毕之后，应该 `pfree` 这个函数的返回值。

参数

`Relation` `rel`

输入关系

返回值

声明关系的名字空间的名字

44.3. 内存管理

Table of Contents

PostgreSQL在内存环境中分配内存， 它提供了在许多地方分配的，有着不同的生命期的许多内存块的一个方便的管理方法。 删除一个环境则释放所有在其内部分配的内存。因此，没必要跟踪独立的对象以避免内存泄漏； 而是只要管理少量的环境。 `palloc` 和相关的函数从"当前" 的环境中分配内存。

`SPI_connect` 创建一个新的内存环境并且将其标记为当前的环境。 `SPI_finish` 恢复前一个内存环境并且删除 `SPI_connect` 创建的环境。这些动作确保在你的过程中分配的临时内存在过程结尾的时候都被回收，避免内存泄漏。

不过，如果你的过程需要返回一个已分配的内存对象(比如一个传递引用的数据类型)， 那么你就不能用 `palloc` 分配返回的对象， 至少是不能在你已经和 SPI 连接上的时候。如果你试图这么做， 那么该对象将在 `SPI_finish` 的时候被释放， 因而你的过程就不能可靠地工作了! 要解决这个问题，使用 `SPI_palloc` 分配内存给你的返回对象。 `SPI_palloc` 从"上层执行者环境" 中分配空间，也就是调用 `SPI_connect` 时候的当前环境内存环境， 该环境是从你的过程返回数值的正确环境。

如果还没有连接到 SPI 的时候调用它， `SPI_palloc` 的行为和简单的 `palloc` 一样。在一个过程和 SPI 管理器连接之前， 当前的内存环境是上层执行者环境，因此所有该过程使用 `palloc` 或者 SPI 工具函数分配的空间都是在这个环境中分配的。

在调用 `SPI_connect` 之后，当前环境是该过程私有的， 由 `SPI_connect` 制作的环境。所有通过 `palloc` 、 `repalloc` 或者 SPI 工具函数(除了 `SPI_copytuple` 、 `SPI_returntuple` 、 `SPI_modifytuple` 和 `SPI_palloc`)分配的内存都是在这个环境中分配的。如果一个过程与 SPI 管理器断开(通过 `SPI_finish`)， 那么当前环境恢复为上层执行器环境， 并且所有在该过程的内存环境中分配的内存都释放掉并且不能再次使用！

所有在本节内描述的函数都可以在已连接的和未连接的过程中使用。 在未连接的过程中，他们的行为和下层的原始后端函数(`palloc` 等)相同。

SPI_palloc

Name

SPI_palloc -- 在上层执行器环境里分配内存

Synopsis

```
void * SPI_palloc(Size size)
```

描述

`SPI_palloc`在上层执行者环境里分配内存。

参数

`Size size`

要分配的存储空间的，以字节记

返回值

指向声明尺寸的新空间的指针

SPI_repalloc

Name

SPI_repalloc -- 在上层执行器环境里重新分配内存

Synopsis

```
void * SPI_repalloc(void * pointer, Size size)
```

描述

`SPI_repalloc` 改变一个前面用 `SPI_palloc` 分配的内存段的大小。

这个函数现在和 `repalloc` 没什么区别。保留它只是为了向后兼容现有的代码。

参数

`void *` `pointer`

一个指向现有存储的指针

`Size` `size`

分配的内存的大小，以字节记

返回值

一个指向新存储空间的指针，大小为你声明的大小，并且从现有区域里拷贝了内存

SPI_pfree

Name

SPI_pfree -- 在上层执行器环境里释放内存

Synopsis

```
void SPI_pfree(void * pointer)
```

描述

`SPI_pfree` 释放前面用 `SPI_palloc` 或 `SPI_repalloc` 分配的内存。

这个函数和 `pfree` 没什么不同。保留它主要是为了向下兼容现有代码。

参数

`void *` `pointer`

一个指向要释放的现有内存的指针

SPI_copytuple

Name

SPI_copytuple -- 在上层执行者环境里制作一个行的拷贝

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

描述

`SPI_copytuple` 在上层执行者环境里制作一个行的拷贝。 这个函数通常用于从触发器里返回一个修改过的行。在声明为返回复合类型的函数里， 应该使用 `SPI_returntuple` 。

参数

`HeapTuple` `row`

要拷贝的行

返回值

拷贝行；只有在 `tuple` 是 `NULL` 的时候为 `NULL`

SPI_returntuple

Name

SPI_returntuple -- 准备把一个行当作 Datum 返回

Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

描述

`SPI_returntuple` 在上层执行者环境里制作一个行的拷贝， 并且把它以行类型 `Datum` 的形式返回。 所返回的指针在返回之前只需要用 `PointerGetDatum` 转换成 `Datum` 。

请注意这个函数应该只用于那些声明为返回复合类型的函数。它不用于触发器；用 `SPI_copytuple` 在触发器中返回一个修改过的行。

参数

`HeapTuple` `row`

将要被拷贝的行

`TupleDesc` `rowdesc`

行的描述符(每次都传递同样的描述符可以获取最高缓冲效率)

返回值

指向拷贝出来的行的 `HeapTupleHeader` ；只有在 `row` 或 `rowdesc` 是 `NULL` 的时候才返回 `NULL`

SPI_modifytuple

Name

SPI_modifytuple -- 通过替换一个给出行的选定的字段创建一行

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

描述

`SPI_modifytuple` 通过给选定的字段替换新值， 拷贝其它字段的原值的方法创建一个新行。不修改输入的行。

参数

`Relation` `rel`

只用于该行的行描述符的源(传递一个关系而不是一个行描述符是一个误特性)。

`HeapTuple` `row`

要修改的行

`int` `ncols`

要修改的字段数目

`int *` `colnum`

长度 `ncols` 的一个数组，包含要修改的字段的数目(字段编号从 1 开始记)

`Datum *` `values`

长度 `ncols` 的一个数组，包含声明字段的新值

`const char *` `nulls`

长度 `ncols` 的一个数组，描述哪个新值是空

如果 `nulls` 是 `NULL` ，那么 `SPI_modifytuple` 假设没有参数为空。否则，如果对应的参数值是非空的，那么 `nulls` 数组的每一项都应该是 `' '`，或者如果对应的参数值为空，那么 `nulls` 数组的每一项都是 `'n'` 。 （在后面这种情况下，对应的 `values` 项中的实际值无关紧要。） 请注意， `nulls` 不是文本字符串，只是一个数组： 它不需要 `'\0'` 终止符。

返回值

修改后的新行，在上层执行者环境里分配；只有在 `row` 是 `NULL` 的时候为 `NULL`

错误时， `SPI_result` 的设置如下：

`SPI_ERROR_ARGUMENT`

如果 `rel` 是 `NULL` ， 或者如果 `row` 是 `NULL` ， 或者如果 `ncols` 小于或者等于 0 ， 或者 `colnum` 是 `NULL` ， 或者如果 `values` 是 `NULL` 。

`SPI_ERROR_NOATTRIBUTE`

如果 `colnum` 包含一个无效的字段编号(小于或者等于 0 或者大于 `row` 里的字段数)

SPI_freetuple

Name

SPI_freetuple -- 释放在上层执行者环境里分配的一行

Synopsis

```
void SPI_freetuple(HeapTuple row)
```

描述

`SPI_freetuple` 释放一个前面在上层执行者环境里分配的行。

这个函数和 `heap_freetuple` 没什么区别。保留它主要是为了向后兼容现有代码。

参数

`HeapTuple row`

要释放的行

SPI_freetuptable

Name

SPI_freetuptable -- 释放一个由 `SPI_execute` 或者类似的函数创建的行集

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

描述

`SPI_freetuptable` 释放一个由前面的 SPI 命令执行函数， 比如 `SPI_execute` 创建的行集。因此，这个函数调用的时候通常用全局变量 `SPI_tuptable` 作为参数。

如果一个 SPI 过程需要执行多条命令并且不想把前面的命令的结果保存到其结尾的话， 那么这个函数就很有用。请注意任何没有释放的行集都会在 `SPI_finish` 的时候释放。另外，如果一个子事务开始并且然后在一个SPI过程的执行中退出， 那么SPI自动释放任何子事务运行时创建的行集。

在PostgreSQL 9.3中开始， `SPI_freetuptable` 包含保护逻辑防卫重复请求删除同一个行集。在以前的版本中，重复删除将会导致崩溃。

参数

`SPITupleTable *` `tuptable`

一个指向要释放的行集的指针，或者什么也不做时为NULL

SPI_freeplan

Name

SPI_freeplan -- 释放一个前面保存的预备语句

Synopsis

```
int SPI_freeplan(SPIPlanPtr plan)
```

描述

`SPI_freeplan` 释放一个以前由 `SPI_prepare` 返回的或者是用 `SPI_keepplan` 或 `SPI_saveplan` 保存的预备语句。

参数

`SPIPlanPtr plan`

指向要释放的语句的指针

返回值

成功时为0；如果 `plan` 是 `NULL` 或无效， 返回 `SPI_ERROR_ARGUMENT`

44.4. 数据改变的可视性

下面规则决定使用 SPI(或者任何其它 C 函数)的函数里面的数据修改的可视性：

- 在一个 SQL 命令执行期间，任何这个命令做的数据改变都是命令本身所看不到的。 比如，在下面命令里：

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  插入的行是 `SELECT` 部分看不到的。

- 命令 C 做的修改可以被 C 之后开始的所有命令看到，不管他们是在 C 里面 (在执行 C 期间)还是在 C 完成后开始的。

- 一个命令，如果是在一个 SQL 命令调用的函数(普通函数或者是触发器)里通过 SPI 执行的， 那么它遵循上面两个规则之一，具体哪个取决于传递给 SPI 的读/写标志。 以只读模式执行的命令遵循第一条规则：它们看不见调用它的命令做的修改。 以读写模式执行的规则遵循第二条规则：它们可以看到迄今为止所做的所有改变。

- 所有标准的过程语言都根据函数的易失性属性设置 SPI 读写模式。 `STABLE` 和 `IMMUTABLE` 函数是以只读模式设置的，而 `VOLATILE` 函数是以读写模式设置的。虽然其它 C 函数可以违反这个规定，但是这么做不是好事。

下一节里包含一些例子，演示了这些规则的应用。

44.5. 例子

这是一个非常简单的 SPI 使用的例子。过程 `execq` 在其第一个参数里接收一个 SQL 命令，第二个参数接收一个行计数， 用 `SPI_exec` 执行这个查询并且返回查询执行过的记录个数。你可以在 `src/test/regress/regress.c` 中的源码树和 spi模块里找更复杂的例子。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /*  把给出的 text 对象转换成 C 字符串*/
    command = text_to_cstring(sql);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*

  * 如果取出了一些行，通过 elog(INFO) 打印它们
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

这个函数使用了调用习惯版本0，为了是让例子更容易理解。在真实的应用里，你应该使用新的版本1的接口。

下面是你在把函数编译成共享库之后声明它的方法（详情请看 Section 35.9.6）：

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS '_filename_'
    LANGUAGE C;
```

下面是一个会话例子：

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- execq插入 0 行
INFO:  EXECQ:  1    -- execq 返回, 被上层 INSERT 插入

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2,就像声明的那样只插入了一行

 execq
-------
     3              -- 10 只是最大值, 3 是真实的行数
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  --  在 (0)+1 里面没有行
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- 在 a+1 里面有一行
(2 rows)

-- 下面示范了改变数据可视性的规则：
```

```
=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                    -- 2行 * 1 (x 在第一行)
 6                    -- 3 行 (2 + 1 刚刚插入的) * 2 (第二行里的 x)
(4 rows)                 ^^^^^^
                         在不同的调用里 execq() 看到的东西不同
```

Chapter 45. 后台工作进程

PostgreSQL可以扩展在分立的进程中运行用户提供的代码。 命令 `postgres` 启动， 停止和监控这些进程，允许它们的生命周期与服务器状态紧密关联。这些进程可以选择连接PostgreSQL的共享内存并与数据库内部连接；它们也可以串行地运行多个事务，就像 常规的客户端连接的服务器进程。另外，通过链接到libpq，它们可以连接到服务器并 且和常规的客户端应用表现一样。

Warning

使用后台工作进程有着相当大的牢固和安全风险。这是因为它们是用C语言写的，有着不受限制的数据 访问方式。乐于使用包含后台工作进程的模块的管理员们应当极度地当心。只有仔细审计过的模块才 应该被允许运行后台工作进程。

只有 `shared_preload_libraries` 里列出的模块能够运行后台工作进程。想要运行后台工作 程序的模块需要通过从它的 `_PG_init()` 调用 `RegisterBackgroundWorker(` BackgroundWorker *worker `)` 来注册 这个程序。 `BackgroundWorker` 结构是这样定义的：

```
typedef void (*bgworker_main_type)(void *main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    bgworker_main_type bgw_main;
    Datum       bgw_main_arg;
} BackgroundWorker;
```

`bgw_name` 是用于日志消息，进程列表和类似环境的一个字符串。

`bgw_flags` 是一个按位与的比特掩码，显示模块想要的容量。 可能的值有 `BGWORKER_SHMEM_ACCESS` （要求访问共享内存）和 `BGWORKER_BACKEND_DATABASE_CONNECTION` （要求能够建立一个数据库连接，通 过这个连接伺候可以运行事务和查询）。一个使用 `BGWORKER_BACKEND_DATABASE_CONNECTION` 连接数据库的后台工作程序还必须用 `BGWORKER_SHMEM_ACCESS` 联接共享内存，否则程序的启动会失败。

`bgw_start_time` 是某种服务器状态，在此期间应当由 `postgres` 启动进程；它可以是下面几个值之一： `BgWorkerStart_PostmasterStart` （ `postgres` 完成自身初始化后就立即启动； 请求此种启动方式的进程不能进行数据库连接）， `BgWorkerStart_ConsistentState` （只 要在一个热备份系统中达到了一致状态就启动，允许进程连接到数据库并运行只读查询）， 以及 `BgWorkerStart_RecoveryFinished` （只要系统进入普通读写状态就启动）。注意在非热备 份系统的服务器中后两个值是作用相当的。注意此设置只在进程将启动时显示；进入不同状态时进程不 停。

`bgw_restart_time` 是以秒记的时间间隔。一旦进程崩溃， `postgres` 应当在重启进程前等待这一段时间。它可以是任何的正值，或者 是 `BGW_NEVER_RESTART` 以表明在进程崩溃时不重启进程。

`bgw_main` 是当进程被启动时指向所运行函数的一个指针。该函数必须使 用 `void *` 类型的单一参数并返回 `void` 类型的值。 `bgw_main_arg` 将作为唯一参数被传递给此函数。注意全局变量 `MyBgworkerEntry` 指向进程注册时传递的 `BackgroundWorker` 结构的一份拷贝。

进程一旦运行，就可以通过调用 `BackgroundWorkerInitializeConnection(` char *dbname* `,` *char* username `)` 连接到一个数据库。这样进程可以使用 `SPI` 接口运行事务和查询。如果 `dbname` 值为NULL，会话不被连接到任何特定数据库，但是可以获取共享的目录。如果 `username` 值为NULL，进程会以 `initdb` 运行时创建的超级用户身份来运行。 每个后台进程只能调用一次BackgroundWorkerInitializeConnection，它不能切换数据库。

Signals are initially blocked when control reaches the 控制 `bgw_main` 函数开始锁定信号，并且必须由此函数解锁；这样在必要时可以允许 进程定制它的信号处理程序。通过调用 `BackgroundWorkerUnblockSignals` 可以在新进程 中为信号解锁，通过调用 `BackgroundWorkerBlockSignals` 可以加锁。

后台工作程序是被预期连续运行的；如果它们干净的退出了， `postgres` 会立即重启它们。 当它们无事可做时，考虑进入可中断的睡眠；通过调用 `WaitLatch()` 可以实现这 一点。调用该函数时确定 `WL_POSTMASTER_DEATH` 标志被设置，并且在 `postgres` 自身终止的紧急情况下为提示的退出验证返回码。

`worker_spi` contrib模块包含一个展示一些有用技术的范例。

VI. 参考手册

这份参考里的条目给各个相关对象提供了权威、完整、正式的概要。 有关使用PostgreSQL的更多信息(叙述、教程、例子)， 可以在本书的其它部分找到。参阅在每个参考页里的交叉引用。

参考手册里的条目也可以在传统的"man"页里获得。

Table of Contents

pg_resetxlog -- 重置一个数据库集群的预写日志以及其它控制内容

postgres -- PostgreSQL 数据库服务器

postmaster -- PostgreSQL 数据库服务器

I. SQL 命令

这部分包含那些PostgreSQL支持的SQL命令的信息。 这里的"SQL"就是该语言通常的含义； 每条命令的标准兼容性信息可以在相关的参考页中找到。

Table of Contents

- ALTER TYPE -- 修改一个类型的定义。
- ALTER USER -- 修改一个数据库角色。
- ALTER USER MAPPING -- 更改用户映射的定义
- ALTER VIEW -- 更改视图定义
- ANALYZE -- 收集与数据库有关的统计信息
- BEGIN -- 开始一个事务块
- CHECKPOINT -- 强制一个事务日志检查点
- CLOSE -- 关闭游标
- CLUSTER -- 根据一个索引对某个表盘簇化排序
- COMMENT -- 定义或者改变一个对象的注释
- COMMIT -- 提交当前事务
- COMMIT PREPARED -- 提交一个早先为两阶段提交准备好的事务
- COPY -- 在表和文件之间拷贝数据
- CREATE AGGREGATE -- 定义一个新的聚集函数
- CREATE CAST -- 定义一个用户定义的转换
- CREATE COLLATION -- 定义一个新的排序规则
- CREATE CONVERSION -- 定义一个新的编码转换
- CREATE DATABASE -- 创建一个新数据库
- CREATE DOMAIN -- 定义一个新域
- CREATE EXTENSION -- 安装一个扩展
- CREATE EVENT TRIGGER -- 定义一个事件触发器
- CREATE FOREIGN DATA WRAPPER -- 定义一个外部数据封装器
- CREATE FOREIGN TABLE -- 定义一个新外部表
- CREATE FUNCTION -- 定义一个新函数
- CREATE GROUP -- 定义一个新数据库角色
- CREATE INDEX -- 创建一个索引
- CREATE LANGUAGE -- define a new procedural language
- CREATE MATERIALIZED VIEW -- 定义一个物化视图
- CREATE OPERATOR -- 定义一个新操作符
- CREATE OPERATOR CLASS -- 定义一个新操作符类
- CREATE OPERATOR FAMILY -- 定义一个新操作符族
- CREATE ROLE -- 定义一个新的数据库角色
- CREATE RULE -- 定义一个新重写规则
- CREATE SCHEMA -- 定义一个新模式
- CREATE SEQUENCE -- 定义一个新序列发生器
- CREATE SERVER -- 定义一个新的外服务器
- CREATE TABLE -- 定义一个新表
- CREATE TABLE AS -- 从一条查询的结果中定义一个新表
- CREATE TABLESPACE -- 定义一个新的表空间
- CREATE TEXT SEARCH CONFIGURATION -- 定义一个新的文本搜索配置

- DROP TEXT SEARCH DICTIONARY -- 删除一个文本搜索字典
- DROP TEXT SEARCH PARSER -- 删除一个文本搜索解析器
- DROP TEXT SEARCH TEMPLATE -- 删除一个文本搜索模板
- DROP TRIGGER -- 删除一个触发器
- DROP TYPE -- 删除一个数据类型
- DROP USER -- 删除一个数据库角色
- DROP USER MAPPING -- 删除用户的外部服务器映射
- DROP VIEW -- 删除一个视图
- END -- 提交当前事务
- EXECUTE -- 执行一个预备语句
- EXPLAIN -- 显示一个语句的执行规划
- FETCH -- 用游标从查询中抓取行
- GRANT -- 赋予访问权限
- INSERT -- 在表中创建新行
- LISTEN -- 监听一个通知
- LOAD -- 加载一个共享库文件
- LOCK -- 锁定一个表
- MOVE -- 定位一个游标
- NOTIFY -- 生成一个通知
- PREPARE -- 创建一个预备语句
- PREPARE TRANSACTION -- 为当前事务做两阶段提交的准备
- REASSIGN OWNED -- 修改数据库对象的属主
- REFRESH MATERIALIZED VIEW -- 替换物化视图的内容
- REINDEX -- 重建索引
- RELEASE SAVEPOINT -- 删除一个先前定义的保存点
- RESET -- 把一个运行时参数重置为缺省值
- REVOKE -- 删除访问权限
- ROLLBACK -- 退出当前事务
- ROLLBACK PREPARED -- 取消一个先前为两阶段提交准备好的事务
- ROLLBACK TO SAVEPOINT -- 回滚到一个保存点
- SAVEPOINT -- 在当前事务里定义一个新保存点
- SECURITY LABEL -- 定义或改变一个应用于对象的安全标签
- SELECT -- 从表或视图中取出若干行
- SELECT INTO -- 从一条查询的结果中定义一个新表
- SET -- 修改运行时参数
- SET CONSTRAINTS -- 设置当前事务的约束检查模式
- SET ROLE -- 在当前会话中设置当前用户标识
- SET SESSION AUTHORIZATION -- 为当前会话设置会话用户标识符和当前用户标识符
- SET TRANSACTION -- 设置当前事务的特性
- SHOW -- 显示运行时参数的值

- START TRANSACTION -- 开始一个事务块
- TRUNCATE -- 清空一个或一组表
- UNLISTEN -- 停止监听通知信息
- UPDATE -- 更新一个表中的行
- VACUUM -- 垃圾收集以及可选地分析一个数据库
- VALUES -- 计算一个或一组行

ABORT

Name

ABORT -- 放弃当前事务

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

描述

`ABORT` 回滚当前事务并且撤销所有当前事务中所做的更改。 这个命令和标准SQL 命令 ROLLBACK 的行为完全一样，只是由于历史原因而保留下来。

参数

`WORK``TRANSACTION`

可选的关键字，它们没有什么作用。

注意

使用 COMMIT 成功地结束一个事务。

在事务外部发出 `ABORT` 不会造成损害，但是会产生一个警告信息。

例子

放弃全部变更：

```
ABORT;
```

兼容性

此命令是 PostgreSQL 基于历史原因做的扩展。等价于标准 SQL 的 `ROLLBACK` 命令。

参见

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

Name

ALTER AGGREGATE -- 修改一个聚集函数的定义

Synopsis

```
ALTER AGGREGATE _name_ ( _argtype_ [ , ... ] ) RENAME TO _new_name_
ALTER AGGREGATE _name_ ( _argtype_ [ , ... ] ) OWNER TO _new_owner_
ALTER AGGREGATE _name_ ( _argtype_ [ , ... ] ) SET SCHEMA _new_schema_
```

描述

`ALTER AGGREGATE` 改变一个聚集函数的定义。

要使用 `ALTER AGGREGATE` ，你必须是该聚集函数的所有者。 要改变一个聚集函数的模式，你必须在新模式上有 `CREATE` 权限。 要改变所有者，你必须是新所有角色的一个直接或间接成员，并且该角色必须在聚集函数的模式上有 `CREATE` 权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建聚集函数不能做的事情。不过，超级用户可以用任何方法任意更改聚集函数的所属关系。）

参数

`_name_`

现有的聚集函数的名称(可以有模式修饰)。

`_argtype_`

聚集函数操作的输入数据类型。要引用一个零参数聚集函数，可以写入 `*` 代替输入数据类型列表。

`_new_name_`

聚集函数的新名字。

`_new_owner_`

聚集函数的新所有者。

`_new_schema_`

聚集函数的新模式。

例子

把一个接受 `integer` 类型参数的聚集函数 `myavg` 重命名为 `my_average` ：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

把一个接受 `integer` 类型参数的聚集函数 `myavg` 的所有者改为 `joe` ：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把一个接受 `integer` 类型参数的聚集函数myavg移动到模式myschema里：

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

兼容性

SQL标准里没有 `ALTER AGGREGATE` 语句。

参见

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

Name

ALTER COLLATION -- 修改一个排序规则定义

Synopsis

```
ALTER COLLATION _name_ RENAME TO _new_name_
ALTER COLLATION _name_ OWNER TO _new_owner_
ALTER COLLATION _name_ SET SCHEMA _new_schema_
```

描述

`ALTER COLLATION` 修改一个排序规则定义.

使用 `ALTER COLLATION` 你必须拥有排序规则. 若要更改拥有者,你必须是这个新所有者角色的直接成员或间接成员, 而且这个角色在 排序规则模式上必须拥有 `CREATE` 权限. (这些限制约束你不能通过放弃或重建排序规则来修改所有者做任何事. 不管怎样,一个超级用户能修改任何排序规则的所有权.)

参数

`_name_`

一个存在的排序规则的名称(可以有模式修饰).

`_new_name_`

排序规则的新名称.

`_new_owner_`

新排序规则的拥有者.

`_new_schema_`

新的排序规则模式.

示例

重命名排序规则 `de_DE` 为 `german`：

```
ALTER COLLATION "de_DE" RENAME TO german;
```

修改排序规则的拥有者 `en_US` 为 `joe`：

```
ALTER COLLATION "en_US" OWNER TO joe;
```

兼容性

在SQL标准中没有 `ALTER COLLATION` 语句.

请参阅

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

Name

ALTER CONVERSION -- 修改编码转换的定义

Synopsis

```
ALTER CONVERSION _name_ RENAME TO _new_name_
ALTER CONVERSION _name_ OWNER TO _new_owner_
ALTER CONVERSION _name_ SET SCHEMA _new_schema_
```

描述

`ALTER CONVERSION` 修改编码转换的定义。

要使用 `ALTER CONVERSION` ，您必须是该编码转换的所有者。要修改其所有者， 您必须是新的所属角色的直接或者间接成员， 并且该角色还必须在该编码转换的模式上有 `CREATE` 权限。（这些限制强制了修改该所有者不会做任何通过删除和重建编码转换不能做的事情。不过，超级用户可以任何方式修改任意编码转换的所有属关系。）

参数

`_name_`

现有的编码转换的名称(可以有模式修饰)。

`_new_name_`

编码转换的新名字。

`_new_owner_`

编码转换的新所有者。

`_new_schema_`

编码转换的新模式名。

例子

要把编码转换 `iso_8859_1_to_utf8` 重新命名为 `latin1_to_unicode` ：

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

要把编码转换 `iso_8859_1_to_utf8` 的所有者改变为 `joe` ：

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

兼容性

SQL 标准里没有 `ALTER CONVERSION` 语句。

参见

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

Name

ALTER DATABASE -- 修改一个数据库

Synopsis

```
ALTER DATABASE _name_ [ [ WITH ] _option_ [ ... ] ]

 这里 `_option_` 可以是以下参数：

CONNECTION LIMIT _connlimit_

ALTER DATABASE _name_ RENAME TO _new_name_

ALTER DATABASE _name_ OWNER TO _new_owner_

ALTER DATABASE _name_ SET TABLESPACE _new_tablespace_

ALTER DATABASE _name_ SET _configuration_parameter_ { TO | = } { _value_ | DEFAULT }
ALTER DATABASE _name_ SET _configuration_parameter_ FROM CURRENT
ALTER DATABASE _name_ RESET _configuration_parameter_
ALTER DATABASE _name_ RESET ALL
```

描述

`ALTER DATABASE` 改变一个数据库的属性。

第一种形式改变某个按数据库设置的相关参数。（见下文细节。） 只有数据库所有者或者超级用户可以改变这些设置。

第二种形式修改该数据库的名称。只有数据库所有者或者超级用户可以重命名一个数据库；非超级用户必须拥有 `CREATEDB` 权限。 当前的数据库不能被重命名。（如果你需要这么做，那么需要先连接另外一个数据库。）

第三种形式改变数据库的所有者。要改变所有者，你必须是该数据库的所有者并且还是新的所有角色的直接或者间接成员， 并且还必须有 `CREATEDB` 权限。 （请注意，超级用户自动拥有所有这些权限。）

第四种形式改变数据库的缺省表空间。要改变缺省表空间，你必须是该数据库的所有者或是超级用户， 并且还必须有新的表空间的读写权限。 这个语句会从物理上将一个数据库原来缺省表空间上的表和索引移至新的表空间。注意不在缺省表空间的表和索引不受此影响。

其他形式为 PostgreSQL 数据库修改缺省的会话运行时配置变量。 任何时候在一个数据库上启动一个新的会话的时候，一些特定的参数值会成为当前会话的缺省值。 与指定数据库相关的缺省会覆盖在 `postgresql.conf` 参数文件中指定的对应值或是从 `postgres` 服务启动时在命令行上指定的参数值。 只有数据库所有者或者超级用户可以为一个数据库修改会话缺省值。有些变量不能用这种方法设置，或者是只能由超级用户设置。

参数

`_name_`

需要修改属性的数据库的名字。

`_connlimit_`

对这个数据库可以做多少个并发连接。-1 意味着没有限制。

`_new_name_`

数据库的新名字。

`_new_owner_`

数据库新的所有者。

`_new_tablespace_`

数据库的新的缺省表空间。

`_configuration_parameter_``_value_`

把数据库的指定配置参数的会话缺省值设置为给定的数值。 如果 `_value_` 是 `DEFAULT` ， 或者是相应的如 `RESET` 选项使用的话，那么与数据库相关的参数设置将被删除，在新的会话中将继承系统级的缺省参数值。 用 `RESET ALL` 可清除所有数据库相关的设置。 用 `SET FROM CURRENT` 可保存会话参数的当前值保存为相关的数据库值。

参阅 SET 和 Chapter 18 获取有关允许的参数名和数值的更多信息。

注意

也可以把一个会话缺省值绑定到一个特定角色上而不是某个数据库上；参阅 ALTER ROLE 。 如果存在冲突，那么角色声明的参数值会覆盖数据库相关的参数值。

例子

要关闭在数据库 `test` 上缺省的索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

兼容性

`ALTER DATABASE` 语句是一个 PostgreSQL 扩展。

参见

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

Name

ALTER DEFAULT PRIVILEGES -- 定义默认的访问权限

Synopsis

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } _target_role_ [, ...] ]
    [ IN SCHEMA _schema_name_ [, ...] ]
    _abbreviated_grant_or_revoke_

where `_abbreviated_grant_or_revoke_` is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`ALTER DEFAULT PRIVILEGES` 允许你设置应用到以后创建的对象的权限. (这不影响已分配给已经存在对象的权限.) 目前, 只有表(包括视图和外部表)、序列、函数 和类型(包括域)的权限能够被更改.

你能修改那些会被你自己或那些你是其中一员的角色所创建的对象的默认权限. 这个权限能被全局设置 (比如：给所有在当前数据库创建的对象), 或者只是给在指定模式中创建的对象. 默认权限规定每个模式的都会被加上，无论全局默认权限是不是为了特殊的对象类型.

根据 GRANT下的解释, 任何对象类型的默认权限通常授权全部可授与权限给对象所有者，以及授权一些权限给 `PUBLIC` . 不管怎样，改行为能通过使用 `ALTER DEFAULT PRIVILEGES` 更改全局默认权限来修改.

参数

`_target_role_`

已经存在并且是一个成员的前角色的名称. 如果 `FOR ROLE` 被省略, 这个当前角色是假设的.

`_schema_name_`

已存在模式的名称. 如果特别指定，随后在该模式里创建的对象，会被修改为默认权限. 如果 `IN SCHEMA` 被省略了,全局默认权限也被更改了.

`_role_name_`

已存在的用来授权或撤销权限的角色的名称. 这个参数，和所有其他在 `_abbreviated_grant_or_revoke_` 中的参数, 作用如 GRANT 或 REVOKE所述, 是给整个类的对象而不是特定的命名对象设置权限.

备注

使用psql's `\ddp` 命令 来获取已存在的默认权限的分配信息. 权限制的含义与GRANT 下 `\dp` 的解释相同。

如果你希望删除一个默认权限已经被修改的角色, 对这个角色来说，撤销默认权限上的修改 或者使用 `DROP OWNED BY` 来摆脱该角色的默认权限默认权限条目，这是必要的.

示例

给你后来在模式 `myschema` 里创建的所有表（和视图）授予SELECT 权限， 并且允许角色 `webuser` 对他们执行INSERT:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

撤销上面的操作, 因此后来创建的表不会拥有比常规情况还多的权限：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

为角色 `admin` 之后创建的所有的函数，移除那些通常授权在函数上的公共EXECUTE 权限:

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

兼容性

在SQL标准中没有 `ALTER DEFAULT PRIVILEGES` 语句.

请参阅

GRANT, REVOKE

ALTER DOMAIN

Name

ALTER DOMAIN -- 修改一个域的定义

Synopsis

```
ALTER DOMAIN _name_
{ SET DEFAULT _expression_ | DROP DEFAULT }
ALTER DOMAIN _name_
{ SET | DROP } NOT NULL
ALTER DOMAIN _name_
ADD _domain_constraint_ [ NOT VALID ]
ALTER DOMAIN _name_
DROP CONSTRAINT [ IF EXISTS ] _constraint_name_ [ RESTRICT | CASCADE ]
ALTER DOMAIN _name_
RENAME CONSTRAINT _constraint_name_ TO _new_constraint_name_
ALTER DOMAIN _name_
VALIDATE CONSTRAINT _constraint_name_
ALTER DOMAIN _name_
OWNER TO _new_owner_
ALTER DOMAIN _name_
RENAME TO _new_name_
ALTER DOMAIN _name_
SET SCHEMA _new_schema_
```

描述

`ALTER DOMAIN` 修改一个域的定义。它有几种子形式：

SET/DROP DEFAULT

这些形式设置或者删除一个域的缺省值。请注意缺省值只适用于随后的 `INSERT` 命令；他们并不影响使用该域已经在表中的行。

SET/DROP NOT NULL

这些形式改变一个域是否标记为允许 NULL 值或者是不允许 NULL 值。 在使用域的字段包含非空的值的时候，你只可以使用 `SET NOT NULL` 。

ADD `_domain_constraint_` [NOT VALID]

这种形式向域中增加一种新的约束，使用的语法和 CREATE DOMAIN 一样。 当一个新的约束增加至域中时，使用这个域的所有列将会按新增的约束条件进行检查。 这些检查也可以通过使用 `NOT VALID` 选项来增加新的约束条件进行关闭； 约束也可以以后通

过 `ALTER DOMAIN ... VALIDATE CONSTRAINT` 语句生效。 新插入或是更新的记录将会按所有约束进行检查，甚至是标记为 `NOT VALID` 的约束。 `NOT VALID` 只是对 `CHECK` 约束生效。

DROP CONSTRAINT [IF EXISTS]

这种形式删除一个域上的约束。如果使用了 `IF EXISTS` 选项并且约束并不存在时，系统不会抛出错误提示， 这种情况下系统只会发出一个提示信息。

RENAME CONSTRAINT

这种形式更改一个域上的约束名称。

VALIDATE CONSTRAINT

这种形式将域上以前以 `NOT VALID` 选项增加的约束进行生效，这样会对域中所有列的数据按这个指定的约束进行验证。

OWNER

这种形式将域的所有者改变为一个指定的用户。

`RENAME`

这种形式将改变域的名称。

SET SCHEMA

这种形式将改变域的模式。所有与这个域有关的约束也会移至新的模式。

要使用 `ALTER DOMAIN` 语句，您必须是该域的所有者。 要修改一个域的模式， 您还必须在新模式上拥有 `CREATE` 权限。 要修改所有者， 您还必须是新的所有角色的直接或间接成员，并且该成员必须在此域的模式上有 `CREATE` 权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建域不能做的事情。不过，超级用户可以以任何方式修改任意域的所有关系。）

参数

`_name_`

一个要修改的现有域的名字(可以有模式修饰)。

`_domain_constraint_`

域的新域约束。

`_constraint_name_`

要删除或是重命名的原有约束名。

`_NOT VALID_`

不对已有数据进行约束的有效性验证。

`CASCADE`

自动级联删除依赖这个约束的对象。

`RESTRICT`

如果有任何依赖对象，则拒绝删除约束。这是缺省行为。

`_new_name_`

域的新名称。

`_new_constraint_name_`

约束的新名称。

`_new_owner_`

域的新的所有者的用户名。

`_new_schema_`

域的新的模式名。

注意

目前，如果命名的域或者任何派生的域用于数据库中任何一个表的一个复合列中， 那么 `ALTER DOMAIN ADD CONSTRAINT` 和 `ALTER DOMAIN SET NOT NULL` 将失败。 最终，他们应该加以改进以达到能够验证这种嵌套列的新约束。

例子

给一个域增加一个 `NOT NULL` 约束：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

从一个域里删除一个 `NOT NULL` 约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

向一个域里增加一个检查约束：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

从一个域里删除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

从一个域里的检查约束重命名：

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

把域移动到另外一个模式：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

兼容性

除了 `OWNER` 、 `RENAME` 、 `SET SCHEMA` 和 `VALIDATE CONSTRAINT` 这些选项是PostgreSQL扩展外， `ALTER DOMAIN` 语句基本与SQL标准兼容。 `ADD CONSTRAINT` 相关的 `NOT VALID` 选项也是PostgreSQL的扩展。

参见

CREATE DOMAIN, DROP DOMAIN

ALTER EXTENSION

Name

ALTER EXTENSION -- 修改扩展定义

Synopsis

```
ALTER EXTENSION _name_ UPDATE [ TO _new_version_ ]
ALTER EXTENSION _name_ SET SCHEMA _new_schema_
ALTER EXTENSION _name_ ADD _member_object_
ALTER EXTENSION _name_ DROP _member_object_

where `_member_object_` is:

  AGGREGATE _agg_name_ (_agg_type_ [, ...] ) |
  CAST (_source_type_ AS _target_type_) |
  COLLATION _object_name_ |
  CONVERSION _object_name_ |
  DOMAIN _object_name_ |
  EVENT TRIGGER _object_name_ |
  FOREIGN DATA WRAPPER _object_name_ |
  FOREIGN TABLE _object_name_ |
  FUNCTION _function_name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] ) |
  MATERIALIZED VIEW _object_name_ |
  OPERATOR _operator_name_ (_left_type_, _right_type_) |
  OPERATOR CLASS _object_name_ USING _index_method_ |
  OPERATOR FAMILY _object_name_ USING _index_method_ |
  [ PROCEDURAL ] LANGUAGE _object_name_ |
  SCHEMA _object_name_ |
  SEQUENCE _object_name_ |
  SERVER _object_name_ |
  TABLE _object_name_ |
  TEXT SEARCH CONFIGURATION _object_name_ |
  TEXT SEARCH DICTIONARY _object_name_ |
  TEXT SEARCH PARSER _object_name_ |
  TEXT SEARCH TEMPLATE _object_name_ |
  TYPE _object_name_ |
  VIEW _object_name_
```

描述

`ALTER EXTENSION` 修改一个已安装的扩展的定义. 这里有几种方式:

`UPDATE`

这种方式更新这个扩展到一个新的版本. 这个扩展必须满足一个适用的更新脚本(或者一系列脚本) 这样就能修改当前安装版本到一个要求的版本.

`SET SCHEMA`

这种方式移动扩展对象到另一个模式. 这个扩展必须*relocatable*才能使命令成功.

`ADD` `_member_object_`

这种方式添加一个已存在对象到扩展. 这主要在扩展更新脚本上有用. 这个对象接着会被视为扩展的成员; 显而易见,该对象只能通过取消扩展来取消 .

`DROP` `_member_object_`

这个方式从扩展上移除一个成员对象. 主要在扩展更新脚本上有用 .这个对象没有被取消, 只是从扩展里分开了.

参考Section 35.15 来获取更多有关这些操作的信息.

你必须拥有扩展来使用 `ALTER EXTENSION` . 这个 `ADD` / `DROP` 方式要求 添加/删除对象的所有权.

参数

`_name_`

已安装扩展的名称.

`_new_version_`

希望的扩展新版本. 这个能被标识符和字面字符重写.如果不是指定的, `ALTER EXTENSION UPDATE` 尝试去更新到不管是什么在扩展的控制文件中显示的默认版本.

`_new_schema_`

给扩展的新模式.

`_object_name_``_agg_name_` `_function_name_` `_operator_name_`

从扩展里被被添加或移除的对象的名称.表, 聚集 , 域, 外链表,函数,操作符, 操作符类, 操作符族,序列, 文本搜索对象,类型, 和能被模式合格的视图的名称.

`_agg_type_`

在聚集函数操作上的一个输入数据类型. 去引用一个零参数聚集函数,写 `*` 代替这些输入数据类型列表.

`_source_type_`

强制转换的源数据类型的名称.

`_target_type_`

强制转换的目标数据类型的名称.

`_argmode_`

这个函数参数的模型: `IN` , `OUT` , `INOUT` , 或者 `VARIADIC` . 如果省略的话, 默认的是 `IN` . 注意 `ALTER EXTENSION` 不关心 `OUT` 参数 , 因为确认函数的一致性只需要输入参数.因此列出 `IN` , `INOUT` , 和 `VARIADIC` 参数就足够了.

`_argname_`

函数参数的名称. 注意 `ALTER EXTENSION` 不关心参数名称, 因为确认函数的一致性只需要参数数据类型.

`_argtype_`

函数参数的数据类型(可以有模式修饰), 如果任何.

`_left_type_` `_right_type_`

操作符的参数的数据类型(可以有模式修饰). 为前缀或后缀运算符的丢失参数写 `NONE` .

`PROCEDURAL`

这是一个干扰词.

示例

更新 `hstore` 扩展到版本 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

更新 `hstore` 扩展的模式为 `utils` :

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

添加一个已存在的函数给 `hstore` 扩展:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

兼容性

`ALTER EXTENSION` 是一个 PostgreSQL扩展.

参阅

CREATE EXTENSION, DROP EXTENSION

ALTER EVENT TRIGGER

Name

ALTER EVENT TRIGGER -- 修改事件触发器的定义

Synopsis

```
ALTER EVENT TRIGGER _name_ DISABLE
ALTER EVENT TRIGGER _name_ ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER _name_ OWNER TO _new_owner_
ALTER EVENT TRIGGER _name_ RENAME TO _new_name_
```

描述

`ALTER EVENT TRIGGER` 改变现有的事件触发器的定义。

你必须是超级用户才能修改事件触发器。

参数

`_name_`

修改现有事件触发器的名字。

`_new_owner_`

事件触发器的新属主的名字。

`_new_name_`

事件触发器的新名字。

`DISABLE` / `ENABLE [ REPLICA | ALWAYS ] TRIGGER`

这些配置触发事件触发器的方式。禁用的触发器是系统已知的，但是当触发事件发生时不执行。 is still known to the system, but is not executed when its triggering 参见 session_replication_role.

兼容性

在SQL规范中没有 `ALTER EVENT TRIGGER` 的声明。

另请参见

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER FOREIGN DATA WRAPPER

Name

ALTER FOREIGN DATA WRAPPER -- 修改外来数据抓取的定义

Synopsis

```
ALTER FOREIGN DATA WRAPPER _name_
    [ HANDLER _handler_function_ | NO HANDLER ]
    [ VALIDATOR _validator_function_ | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] _option_ ['_value_'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER _name_ OWNER TO _new_owner_
ALTER FOREIGN DATA WRAPPER _name_ RENAME TO _new_name_
```

Description

`ALTER FOREIGN DATA WRAPPER` 修改外来数据抓取的定义。 这个命令的第一个形式是改变函数的支持和外来数据抓取（至少要求一个语句）的属性选项。 第二个形式是改变外来抓取数据的所有者。

只有超级用户能够修改外来抓取数据。 此外，只有超级用户能够。

Parameters

`_name_`

已有外来数据抓取的名字。

`HANDLER` `_handler_function_`

为外来数据抓取指定一个新的处理函数。

`NO HANDLER`

这个参数用来指定外来数据抓取不再拥有处理函数。

注意，使用外来数据抓取但没有handler的外表不能访问。

`VALIDATOR` `_validator_function_`

为外来数据抓取指定一个新的验证函数。

注意，在修改验证器选项后外来数据抓取，服务端和用户映射会失效。 用户在使用外来数据抓取之前需要保证这个选项是正确的。

`NO VALIDATOR`

这个用来指定外来数据抓取不再有验证器函数。

`OPTIONS ( [ ADD | SET | DROP ]` `_option_` [' `_value_` '] [, ...])

修改外来数据抓取的选项。 `ADD` , `SET` , and `DROP` 指定表现的动作。 `ADD` 假定如果没有明确指定操作。选项名必须唯一； 名字和指也要证实使用外来数据抓取的验证器函数。

`_new_owner_`

外来数据抓取新的所有者的用户名。

`_new_name_`

外来数据抓取的新名称。

例子

修改一个外来数据 `dbi` , 增加选项 `foo` , drop `bar` :

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

修改外来数据抓取 `dbi` 验证器为 `bob.myvalidator` :

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Compatibility

`ALTER FOREIGN DATA WRAPPER` 确认为 ISO/IEC 9075-9 (SQL/MED), 排除 `HANDLER` , `VALIDATOR` , `OWNER TO` , 和 `RENAME` 子句是扩展。

See Also

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

Name

ALTER FOREIGN TABLE -- 修改外部表的定义

Synopsis

```
ALTER FOREIGN TABLE [ IF EXISTS ] _name_
    _action_ [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] _name_
    RENAME [ COLUMN ] _column_name_ TO _new_column_name_
ALTER FOREIGN TABLE [ IF EXISTS ] _name_
    RENAME TO _new_name_
ALTER FOREIGN TABLE [ IF EXISTS ] _name_
    SET SCHEMA _new_schema_
这里 `_action_` 是下列之一:

    ADD [ COLUMN ] _column_name_ _data_type_ [ COLLATE _collation_ ] [ _column_constraint
    DROP [ COLUMN ] [ IF EXISTS ] _column_name_ [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] _column_name_ [ SET DATA ] TYPE _data_type_
    ALTER [ COLUMN ] _column_name_ SET DEFAULT _expression_
    ALTER [ COLUMN ] _column_name_ DROP DEFAULT
    ALTER [ COLUMN ] _column_name_ { SET | DROP } NOT NULL
    ALTER [ COLUMN ] _column_name_ SET STATISTICS _integer_
    ALTER [ COLUMN ] _column_name_ SET ( _attribute_option_ = _value_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ RESET ( _attribute_option_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ OPTIONS ( [ ADD | SET | DROP ] _option_ ['_value_'] [,
    OWNER TO _new_owner_
    OPTIONS ( [ ADD | SET | DROP ] _option_ ['_value_'] [, ... ])
```

描述

`ALTER FOREIGN TABLE` 修改一个外部表的定义。这里有好几种形式:

`ADD COLUMN`

这种形式为外部表增加一个字段，语法同CREATE FOREIGN TABLE。 与向常规表增加字段不同，外部表增加字段不会引起存储空间的变化，这种操作简单地声明了外部表是可获得这些新字段。

`DROP COLUMN [ IF EXISTS ]`

这种形式删除外部表的一个字段。如果表中的该字段被该表之外的其他对象依赖， 那就需要增加 `CASCADE` ；比如依赖该字段的视图。 如果指定了 `IF EXISTS` ，则在字段不存在时也不会报错,只是产生一个注意信息。

`IF EXISTS`

如果外部表不存在，不会报错，而是产生一个注意信息。

`SET DATA TYPE`

这种形式修改一个外部表字段的数据类型。

`SET` / `DROP DEFAULT`

这种形式为一个字段设置或者删除缺省值。缺省值只应用于随后的 `INSERT` 或 `UPDATE` 命令；它们不会导致已经在表中的数值的修改。

`SET` / `DROP NOT NULL`

标志一个字段是否允许为空值。

`SET STATISTICS`

这个形式为随后的ANALYZE操作设置每字段的统计收集目标。 更多细节请参考类似形式的ALTER TABLE

`SET (` `_attribute_option_` = `_value_` [, ...]) `RESET (` `_attribute_option_` [, ...])

这种形式设置或重设每一个属性选项，更多细节，参考ALTER TABLE

`OWNER`

这个形式改变外部表的所有者为指定所有者

`RENAME`

`RENAME` 形式改变一个外部表的名字或者是外部表中一个独立字段的名字。

`SET SCHEMA`

这种形式把外部表从一个模式移植到另一个模式。

`OPTIONS ( [ ADD | SET | DROP ]` `_option_` [' `_value_` '] [, ...])

改变外部表或者外部表字段的选项。 `ADD` , `SET` , 和 `DROP` 指定执行的操作。 如果没有显式设置操作，那么默认就是ADD。选项的名字不允许重复(尽管表选项和表字段选项可以有相同的名字)。 选项的名称和值也会通过外部数据封装器的类库库进行校验。

除 `RENAME` 和 `SET SCHEMA` 之外的操作都可以写组合在一起，同时执行。 例如，在单一一条命令中，可以添加多个列和（或者）修改多个列的数据类型。

你必须是表的所有者才能使用 `ALTER FOREIGN TABLE` 命令。 要修改外部表的模式，必须在被修改的新模式中拥有 `CREATE` 权限。 要改变外部表的所有权，你必须是新角色的直接或间接成员，并且这个新角色必须在该表的模式上具有 `CREATE` 权限。 （这强制限制在进行更改所有者的操作时，如果无法通过删除后重建表的方式完成，那么就不能做任何事。不过，超级用户可以用任何方法任意更改表的所有者。） 要增加一个字段或修改一个字段类型，必须在数据类型上有 `USAGE` 权限

参数

`_name_`

待修改的已存在外部表的名字（可以用模式修饰）。

`_column_name_`

现存或新的字段名称。

`_new_column_name_`

字段的新名称

`_new_name_`

表的新名称

`_data_type_`

新字段的类型，或者现存字段的新类型。

`CASCADE`

自动删除依赖于被删除字段的对象 （比如，引用该字段的视图）。

`RESTRICT`

如果有依赖于此字段的对象，则拒绝删除该字段。这是缺省行为。

`_new_owner_`

该表的新所有者的用户名。

`_new_schema_`

表将要移植到的新的模式名称

注意

关键字 `COLUMN` 是多余的，可以省略。

通过 `ADD COLUMN` 添加字段，或通过 `DROP COLUMN` 删除字段， 或通过添加 `NOT NULL` 进行约束，或通过 `SET DATA TYPE` 修改字段的数据类型， 都不会检查数据一致性。确保表定义和远端表定义相匹配是用户的责任。

参考CREATE FOREIGN TABLE 部分获取更多有效参数的描述。

示例

设定一个字段不为空:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

修改外部表的选项：

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2, 'value2',
```

兼容性

`ADD`， `DROP` 和 `SET DATA TYPE` 这三种形式符合SQL标准。 其他形式是PostgreSQL的扩展。另外，一个 `ALTER FOREIGN TABLE` 命 令中设置多个操作的功能也是扩展。

`ALTER FOREIGN TABLE DROP COLUMN` 可以删除外部表中的仅有的唯一一个列， 删除之后就是一个无字段的表。这个特性是PostgreSQL的扩展，标准SQL中不允许字段的表。

ALTER FUNCTION

Name

ALTER FUNCTION -- 修改一个函数的定义

Synopsis

```
ALTER FUNCTION _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] )
_action_ [ ... ] [ RESTRICT ]
ALTER FUNCTION _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] )
RENAME TO _new_name_
ALTER FUNCTION _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] )
OWNER TO _new_owner_
ALTER FUNCTION _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] )
SET SCHEMA _new_schema_
where `_action_` is one of:
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
COST _execution_cost_
ROWS _result_rows_
SET _configuration_parameter_ { TO | = } { _value_ | DEFAULT }
SET _configuration_parameter_ FROM CURRENT
RESET _configuration_parameter_
RESET ALL
```

描述

`ALTER FUNCTION` 修改一个函数的定义。

要使用 `ALTER FUNCTION` ，你必须是该函数的所有者。 要修改一个函数的模式，你还必须在新模式上拥有 `CREATE` 权限。 要修改所有者，你必须还是新的所有角色的直接或者间接的成员，并且该成员必须在此函数的模式上有 `CREATE` 权限。 （这些限制保证了修改所有者和删除、重建函数的动作没啥区别。 不过，超级用户可以用任何方法修改函数的所有关系。）

参数

`_name_`

一个现有的函数名字(可以有模式修饰）。

`_argmode_`

参数的模式： `IN` 、 `OUT` 、 `INOUT` 或是 `VARIADIC` 。 如省略的话，缺省是 `IN` 。 注意 `ALTER FUNCTION` 实际不会关注任何 `OUT` 参数， 因为确认函数的逻辑只需要知道输入参数。 因此列出 `IN` 、 `INOUT` 和 `VARIADIC` 参数就足够了。

`_argname_`

参数的名字。 请注意 `ALTER FUNCTION` 实际上不会关注参数的名字，因为只有参数的数据类型用于确认函数的逻辑。

`_argtype_`

如果有的话，是该函数参数的数据类型(可以用模式修饰)。

`_new_name_`

函数的新名字。

`_new_owner_`

函数的新所有者。请注意如果函数标记为 `SECURITY DEFINER` ，那么它随后将以新的所有者执行。

`_new_schema_`

函数的新模式名称。

`CALLED ON NULL INPUT``RETURNS NULL ON NULL INPUT` `STRICT`

`CALLED ON NULL INPUT` 选项会在函数的部分或是全部的参数是NULL的时候会调用它。 `RETURNS NULL ON NULL INPUT` 或 `STRICT` 把函数改成如果任何一个参数是NULL就根本不执行并且自动返回NULL 。 参阅CREATE FUNCTION获取更多信息。

`IMMUTABLE` `STABLE` `VOLATILE`

把函数的易失属性修改为指定类型。参阅CREATE FUNCTION了解更多细节。

`[ EXTERNAL ] SECURITY INVOKER` `[ EXTERNAL ] SECURITY DEFINER`

`SECURITY INVOKER` 表明该函数将带着调用它的用户的权限执行。 `SECURITY DEFINER` 声明该函数将以创建它的用户的权限执行。 关键字 `EXTERNAL` 不是必需的，仅是为了和 SQL 兼容。参阅 CREATE FUNCTION 了解更多细节。

`LEAKPROOF`

更改函数的密封性。 参阅 CREATE FUNCTION 了解更多细节。

`COST` `_execution_cost_`

更改函数的估计执行成本。 参阅CREATE FUNCTION了解更多细节。

`ROWS` `_result_rows_`

更改返回数据集类型函数的返回估计行数。参阅CREATE FUNCTION了解更多细节。

`_configuration_parameter_` `_value_`

当函数被调用时，添加或者更改对配置参数值的配置。如果 `_value_` 是 `DEFAULT` ， 或者相应地 `RESET` 选项被使用，函数的局部设置参数会被清除，这样函数会使用当前环境中的参数值来执行。 使用 `RESET ALL` 来清理所有的函数局部设置参数。 `SET FROM CURRENT` 保存会话的参数当前值为函数调用时会应用的参数值。

参阅SET和Chapter 18 了解更多关于允许的参数名称和数值的信息。

`RESTRICT`

可忽略的选项，仅为了符合SQL标准。

例子

把参数类型为 `integer` 的函数 `sqrt` 重命名为 `square_root` ：

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

把参数类型为 `integer` 的函数 `sqrt` 的所有者修改为 `joe` ：

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

把参数类型为 `integer` 的函数 `sqrt` 的模式修改为 `maths` ：

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

调整自动为函数设定的搜索路径：

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

禁用针对一个函数的 `search_path` 自动设置：

```
ALTER FUNCTION check_password(text) RESET search_path;
```

这个函数在执行时所用的搜索路径将是根据调用者所用的搜索路径。

兼容性

这个语句部分兼容SQL标准里面的`ALTER FUNCTION`语句。 标准允许修改函数的更多属性，但是没有提供修改函数名字的功能， 也没有提供把一个函数定义为安全定义器的功能， 更没有修改函数所有者、模式、易失性的功能。 标准还要求`RESTRICT`关键字，这在PostgreSQL里是可选的。

参见

CREATE FUNCTION, DROP FUNCTION

ALTER GROUP

Name

ALTER GROUP -- 修改角色名或者成员关系

Synopsis

```
ALTER GROUP _group_name_ ADD USER _user_name_ [, ... ]
ALTER GROUP _group_name_ DROP USER _user_name_ [, ... ]
ALTER GROUP _group_name_ RENAME TO _new_name_
```

描述

`ALTER GROUP`修改一个用户组的属性。 这是一条过时的命令，不过出于向后兼容的原因，还被接受， 因为组（以及用户）都已经被更一般的概念：角色，给代替了。

前两个形式从组中增加或者删除用户。 （任何角色都可以当作"用户"或者"组"来做这个事情。） 这个变体实际上等效于给命名为"组"的角色名上赋予或者撤销成员关系； 因此，做这个事情的比较好的方法是GRANT 或者REVOKE。

第三种变体修改一个组的名字。它完全等效于用 ALTER ROLE 重命名角色。

参数

`_group_name_`

要更改的组（角色）名称。

`_user_name_`

准备向组（角色）中增加或从组（角色）中删除的用户名。 用户必须已经存在。 `ALTER GROUP` 并不创建或删除用户。

`_new_name_`

组的新名字。

例子

向组中增加用户：

```
ALTER GROUP staff ADD USER karl, john;
```

从组中删除用户：

```
ALTER GROUP workers DROP USER beth;
```

兼容性

SQL标准里没有 `ALTER GROUP` 语句。

参见

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

Name

ALTER INDEX -- 改变一个索引的定义

Synopsis

```
ALTER INDEX [ IF EXISTS ] _name_ RENAME TO _new_name_
ALTER INDEX [ IF EXISTS ] _name_ SET TABLESPACE _tablespace_name_
ALTER INDEX [ IF EXISTS ] _name_ SET ( _storage_parameter_ = _value_ [, ... ] )
ALTER INDEX [ IF EXISTS ] _name_ RESET ( _storage_parameter_ [, ... ] )
```

描述

`ALTER INDEX` 改变一个现有索引的定义。 它有几种子形式：

`IF EXISTS`

使用此选项，当索引不存在时不会抛出错误，只会有一个提示信息。

`RENAME`

`RENAME` 只改变索引的名字。对存储的数据没有影响。

`SET TABLESPACE`

这个选项会改变索引的表空间为指定表空间，并且把索引相关的数据文件移动到新的表空间里。 参见CREATE TABLESPACE

`SET (` `_storage_parameter_` = `_value_` [, ...])

这个选项会改变索引的一个或多个索引方法特定的存储参数。参见CREATE INDEX获取可用参数的细节。 需要注意的是索引内容不会被这个命令立即修改，根据参数的不同， 你可能需要使用REINDEX重建索引来获得期望的效果。

`RESET (` `_storage_parameter_` [, ...])

这个选项重置索引的一个或多个索引方法特定的存储参数为缺省值。 与 `SET` 一样，可能需要使用 `REINDEX` 来完全更新索引。

参数

`_name_`

要修改的索引的名字(可以有模式修饰)。

`_new_name_`

索引的新名字。

`_tablespace_name_`

索引将移动到的表空间的名字。

`_storage_parameter_`

索引方法特定的存储参数的名字。

`_value_`

索引方法特定的存储参数的新值。根据参数的不同，这可能是一个数字或单词。

注意

这些操作也可以用ALTER TABLE进行。 `ALTER INDEX` 实际上只是 `ALTER TABLE` 用于索引的形式的一个别名。

以前还有一个 `ALTER INDEX OWNER` 变种，但是现在忽略了(带一个警告)。 一个索引不能有一个和其表的属主不同的所有者。改变该表的所有者将自动改变索引的所有者。

不允许修改系统表索引的任何部分。

例子

重命名一个现有的索引：

```
ALTER INDEX distributors RENAME TO suppliers;
```

把一个索引移动到另外一个表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

改变索引的占位因数(假定该索引方法支持它)：

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

兼容性

`ALTER INDEX`是PostgreSQL扩展。

参见

CREATE INDEX, REINDEX

ALTER LANGUAGE

Name

ALTER LANGUAGE -- 修改一个过程语言的定义

Synopsis

```
ALTER [ PROCEDURAL ] LANGUAGE _name_ RENAME TO _new_name_
ALTER [ PROCEDURAL ] LANGUAGE _name_ OWNER TO _new_owner_
```

描述

`ALTER LANGUAGE` 修改一个语言的定义。 目前唯一的功能就是重命名语言或是设置新的所有者。 只有超级用户或者语言的所有者 可以使用 `ALTER LANGUAGE` 。

参数

`_name_`

语言的名字。

`_new_name_`

语言的新名字。

`_new_owner_`

语言的新的所有者。

兼容性

SQL标准里没有 `ALTER LANGUAGE` 语句。。

参见

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

Name

ALTER LARGE OBJECT -- change the definition of a large object

Synopsis

```
ALTER LARGE OBJECT _large_object_oid_ OWNER TO _new_owner_
```

描述

`ALTER LARGE OBJECT` 更改一个large object的定义。它的唯一的功能是分配一个新的所有者。使用 `ALTER LARGE OBJECT` 必须是超户或者是其所有者。

参数

`_large_object_oid_`

要被变large object的OID

`_new_owner_`

large object新的所有者

兼容性

在SQL标准里没有 `ALTER LARGE OBJECT` 语句。

相关内容

Chapter 32

ALTER MATERIALIZED VIEW

Name

ALTER MATERIALIZED VIEW -- 修改物化视图的定义

Synopsis

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] _name_
    _action_ [, ... ]
ALTER MATERIALIZED VIEW [ IF EXISTS ] _name_
    RENAME [ COLUMN ] _column_name_ TO _new_column_name_
ALTER MATERIALIZED VIEW [ IF EXISTS ] _name_
    RENAME TO _new_name_
ALTER MATERIALIZED VIEW [ IF EXISTS ] _name_
    SET SCHEMA _new_schema_

where `_action_` is one of:

    ALTER [ COLUMN ] _column_name_ SET STATISTICS _integer_
    ALTER [ COLUMN ] _column_name_ SET ( _attribute_option_ = _value_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ RESET ( _attribute_option_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    CLUSTER ON _index_name_
    SET WITHOUT CLUSTER
    SET ( _storage_parameter_ = _value_ [, ... ] )
    RESET ( _storage_parameter_ [, ... ] )
    OWNER TO _new_owner_
    SET TABLESPACE _new_tablespace_
```

描述

`ALTER MATERIALIZED VIEW`改变一个现有物化视图的各种辅助属性。

要使用`ALTER MATERIALIZED VIEW`，你必须拥有该物化视图。要改变一个物化视图的模式， 也必须有在新模式上的`CREATE`权限。要修改所有者，你必须是新所有角色的直接或间接成员，并且该角色必须在该物化视图的模式上拥有`CREATE`权限。 （通过删除然后重建物化视图，这些限制强制修改所有者不做任何你不能做的事情。不过， 一个超级用户可以修改任意视图的所有权。）

`ALTER MATERIALIZED VIEW`可用的语句的从属形式和动作是`ALTER TABLE`可用的一个子集，当用于物化视图时有相同的含义。 参阅ALTER TABLE的描述获取细节。

参数

`_name_`

现有物化视图的名字（可以有模式修饰）。

`_column_name_`

一个新的或现有字段的名字。

`_new_column_name_`

为一个现有字段新增名字。

`_new_owner_`

物化视图新的所有者的用户名。

`_new_name_`

物化视图的新名字。

`_new_schema_`

物化视图的新模式。

例子

重命名物化视图 `foo` 为 `bar`：

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

兼容性

`ALTER MATERIALIZED VIEW` 是一个PostgreSQL扩展。

又见

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

Name

ALTER OPERATOR -- 修改一个操作符的定义

Synopsis

```
ALTER OPERATOR _name_ ( { _left_type_ | NONE } , { _right_type_ | NONE } ) OWNER TO _new_
ALTER OPERATOR _name_ ( { _left_type_ | NONE } , { _right_type_ | NONE } ) SET SCHEMA _ne
```

描述

`ALTER OPERATOR`改变一个操作符的定义。 目前唯一能用的功能是改变操作符的所有者。

要使用ALTER OPERATOR，你必须是该操作符的所有者。 要修改所有者，你还必须是新的所有角色的直接或间接成员，并且该成员必须在此操作符的模式上有`CREATE`权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建操作符不能做的事情。不过，超级用户可以以任何方式修改任意操作符的所有权。）

参数

`_name_`

一个现有操作符的名字(可以有模式修饰)。

`_left_type_`

操作符的左操作数的数据类型；如果没有左操作数，那么写`NONE`。

`_right_type_`

操作符的右操作数的数据类型；如果没有右操作数，那么写`NONE`。

`_new_owner_`

操作符的新所有者。

`_new_schema_`

操作符的新模式名。

例子

改变一个用于 `text` 的用户定义操作符a @@ b：

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

兼容性

SQL标准里没有 `ALTER OPERATOR` 语句。

参见

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

Name

ALTER OPERATOR CLASS -- 修改一个操作符类的定义

Synopsis

```
ALTER OPERATOR CLASS _name_ USING _index_method_ RENAME TO _new_name_
ALTER OPERATOR CLASS _name_ USING _index_method_ OWNER TO _new_owner_
ALTER OPERATOR CLASS _name_ USING _index_method_ SET SCHEMA _new_schema_
```

描述

`ALTER OPERATOR CLASS`修改一个操作符类的定义。。

要使用`ALTER OPERATOR CLASS`，你必须该操作符类的所有者。 要修改所有者，你还必须是新的所有角色的直接或间接成员，并且该成员必须在此操作符类的模式上有`CREATE`权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建操作符类不能做的事情。不过，超级用户可以以任何方式修改任意操作符类的所有权。）

参数

`_name_`

一个现有操作符类的名字(可以有模式修饰)。

`_index_method_`

一个操作符类操作的索引方法的名字。

`_new_name_`

操作符类的新名字。

`_new_owner_`

操作符类的新所有者。

`_new_schema_`

操作符类的新模式名。

兼容性

SQL标准里没有 `ALTER OPERATOR CLASS` 语句。

参见

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

Name

ALTER OPERATOR FAMILY -- 修改操作符族的定义

Synopsis

```
ALTER OPERATOR FAMILY _name_ USING _index_method_ ADD
  {  OPERATOR _strategy_number_ _operator_name_ ( _op_type_, _op_type_ ) [ FOR SEARCH | F
   | FUNCTION _support_number_ [ ( _op_type_ [ , _op_type_ ] ) ] _function_name_ ( _argum
  } [, ... ]
ALTER OPERATOR FAMILY _name_ USING _index_method_ DROP
  {  OPERATOR _strategy_number_ ( _op_type_ [ , _op_type_ ] )
   | FUNCTION _support_number_ ( _op_type_ [ , _op_type_ ] )
  } [, ... ]
ALTER OPERATOR FAMILY _name_ USING _index_method_ RENAME TO _new_name_
ALTER OPERATOR FAMILY _name_ USING _index_method_ OWNER TO _new_owner_
ALTER OPERATOR FAMILY _name_ USING _index_method_ SET SCHEMA _new_schema_
```

描述

`ALTER OPERATOR FAMILY` 修改一个操作符族的定义。 你可以添加操作符和支持函数到该族、从该族中删除它们或修改该族的名字或所有者。

当使用 `ALTER OPERATOR FAMILY` 添加操作符和支持函数到一个族时， 它们不是该操作符族中任意指定操作符类的一部分，只是"松散"在该族中。 这表明这些操作符和函数与该族的语义兼容，但不是正确运行任何指定索引的所需。 （操作符和函数要成为索引的所需，应该被声明为一个操作符类的一部分；参阅 CREATE OPERATOR CLASS。）PostgreSQL 将允许一个族的松散成员在任何时候被从该族中删除，但是操作符类的成员不能被删除， 除非删除整个类和任何依赖于它的索引。典型的，单数据类型操作符和函数是操作符类的一部分， 因为需要它们支持一个特定数据类型的索引，而交叉数据类型操作符和函数由族的松散成员组成。

要使用 `ALTER OPERATOR FAMILY`， 你必须是一个超级用户。 （做这个限制是因为一个错误的操作符族定义会混淆或者甚至崩溃服务器。）

`ALTER OPERATOR FAMILY` 目前并不检查操作符族定义是否包括索引方法所需的所有操作符和函数， 也不检查操作符和函数是否来自一个自相一致的集合。定义一个有效的操作符族是用户的责任。

参阅Section 35.14获取更多信息。

参数

`_name_`

一个现有操作符族的名字（可以有模式修饰）。

`_index_method_`

使用这个操作符族的索引方法的名字。

`_strategy_number_`

该索引方法的与该操作符族相关的一个操作符的策略数。

`_operator_name_`

与该操作符族相关的一个操作符的名字（可以有模式修饰）。

`_op_type_`

在一个 `OPERATOR` 子句中，该操作符的操作数的数据类型或 `NONE` 表示一个左目或右目操作符。不像 `CREATE OPERATOR CLASS` 中可比较的语法，必须总是指定操作数的数据类型。

在一个 `ADD FUNCTION` 子句中，函数的操作数的数据类型如果与该函数的输入数据类型不同，则打算支持操作数的数据类型。对于B-tree比较函数和哈希函数，不需要指定 `_op_type_`，因为函数的输入数据类型总是要使用的正确类型。对于B-tree排序支持函数和所有在GiST、SP-GiST和GIN操作符类中的函数，必须指定该函数要使用的操作数的数据类型。

在 `DROP FUNCTION` 子句中，打算支持的函数的操作数的数据类型必须指定。

`_sort_family_name_`

描述与一个排序操作符相关的排序顺序的现有 `btree` 操作符族的名字 （可以有模式修饰）。

如果既没有指定 `FOR SEARCH` 也没有指定 `FOR ORDER BY`， 那么 `FOR SEARCH` 是缺省。

`_support_number_`

与该操作符族相关的一个函数的索引方法的支持过程数量。

`_function_name_`

该操作符族的索引方法支持过程的函数的名字（可以有模式修饰）。

`_argument_type_`

该函数的参数数据类型。

`_new_name_`

操作符族的新名字。

`_new_owner_`

操作符族的新所有者。

`_new_schema_`

操作符族的新模式。

`OPERATOR` 和 `FUNCTION` 子句可以以任意顺序出现。

注意

请注意， `DROP` 语法只指定了操作符族中的"位置"， 通过策略或支持数和输入数据类型。占领该位置的操作符或函数的名字没有提及。 另外，对于 `DROP FUNCTION` 来说，要指定的类型是该函数打算支持的输入数据类型；对于GiST、SP-GiST和GIN索引来说，可能与该函数的实际输入参数类型无关。

因为索引机制在使用函数之前并不检查函数上的访问权限， 包括一个操作符族中的函数或操作符相当于赋予了公共执行权限。 这对于操作符族中有用的函数的种类通常不是一个问题。

操作符不应该通过SQL函数定义。一个SQL函数可以内联到调用查询， 这将阻止优化器认识到该查询匹配一个索引。

在PostgreSQL 8.4之前， `OPERATOR` 子句 会包括一个 `RECHECK` 选项。现在不再支持这个了，因为一个索引操作符是否是 "松散的"决定了运行时的动态。这允许有效的处理操作符可能或可能不是松散的的情况。

例子

下列的示例命令添加了跨数据类型的操作符和支持函数到一个操作符族， 该操作符族早已包含数据类型 `int4` 和 `int2` 的B-tree操作符类。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

再次删除这些条目：

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

兼容性

SQL标准中没有`ALTER OPERATOR FAMILY`语句。

又见

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER ROLE

Name

ALTER ROLE -- 修改一个数据库角色

Synopsis

```
ALTER ROLE _name_ [ [ WITH ] _option_ [ ... ] ]

这里的 `_option_` 可以是：

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT _connlimit_
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD '_password_'
| VALID UNTIL '_timestamp_'

ALTER ROLE _name_ RENAME TO _new_name_

ALTER ROLE _name_ [ IN DATABASE _database_name_ ] SET _configuration_parameter_ { TO | =
ALTER ROLE { _name_ | ALL } [ IN DATABASE _database_name_ ] SET _configuration_parameter_
ALTER ROLE { _name_ | ALL } [ IN DATABASE _database_name_ ] RESET _configuration_paramete
ALTER ROLE { _name_ | ALL } [ IN DATABASE _database_name_ ] RESET ALL
```

描述

`ALTER ROLE`修改一个PostgreSQL角色的属性。

这个命令的第一种形式可以修改很多CREATE ROLE里面声明的角色属性。（除了增加和删除成员关系的选项之外，所有可能的属性都有介绍；使用GRANT和REVOKE可以实现前述两件事。）没有在命令里提到的属性维持它们以前的设置。数据库超级用户可以给任何角色改变任何设置。拥有`CREATEROLE`权限的角色可以修改任意这些设置，但是只能给非超级用户和非复制用户角色设置。普通的角色只能修改它们自己的口令。

第二种形式可以修改角色的名称。数据库超级用户可以修改任何角色的名称。拥有`CREATEROLE`权限的角色可以给非超级用户角色进行重命名。当前会话的用户的角色是不能改名的。（如果一定需要这么做，则必须以另外一个用户的身份连接系统。）因为`MD5`加密的口令使用角色名字作为加密的盐粒，所以，如果口令是`MD5`加密的，那么给一个角色改名会清空其口令。

其他的形式更改一个角色的会话配置参数默认值，该值要么针对所有的数据库， 要么使用 `IN DATABASE` 选项，仅针对指定的数据库。 如果是未指定角色名称而是使用了 `ALL` 选项，则所有角色的相关参数都会更改。 当同时使用 `ALL` 和 `IN DATABASE` 选项时，就等同于使用 `ALTER DATABASE ... SET ...` 。

当角色随后开启一个新会话，指定的值变成了会话的默认值，这些参数值会覆盖任何 `postgresql.conf` 中的设置或者从 `postgres` 命令行接收到的参数值。 这仅在登录时发生；执行SET ROLE或者SET SESSION AUTHORIZATION不会引起新的配置值设置。 为所有数据库所设定的参数会被附加到一个角色上的特定数据库的参数所覆盖。为指定数据库或指定角色所设定的参数会覆盖为所有角色设定的参数。

超级用户可以更改任何一个会话默认值。有 `CREATEROLE` 权限的角色可以为非超级用户角色更改默认值。 普通的角色只能为自己设置默认值。 某些配置变量不能这样设置，或者只要超级用户才能执行。 只有超级用户才能为所有数据库中的所有角色更改参数设置。

参数

`_name_`

需要修改属性的角色的名称。

`SUPERUSER``NOSUPERUSER` `CREATEDB` `NOCREATEDB` `CREATEROLE` `NOCREATEROLE` `CREATEUSER` `NOCREATEUSER` `INHERIT` `NOINHERIT` `LOGIN` `NOLOGIN` `REPLICATION` `NOREPLICATION` `CONNECTION LIMIT` `_connlimit_` `PASSWORD` `_password_` `ENCRYPTED` `UNENCRYPTED` `VALID UNTIL` ' `_timestamp_` '

这些选项修改由CREATE ROLE初始设置的属性。 要获取更多详细信息，请参阅 `CREATE ROLE` 参考页。

`_new_name_`

角色的新名字。

`_database_name_`

要设置配置变量的数据库的名称。

`_configuration_parameter_` `_value_`

把该角色指定的参数缺省值设置为给定值。如果 `_value_` 是 `DEFAULT` 或是相当于使用 `RESET` ， 指定角色的参数值会被清除，这样该角色将在新的会话里继承系统级的参数缺省值。 使用 `RESET ALL` 会清除所有角色相关的设置。 使用 `SET FROM CURRENT` 会将会话的当前参数值保存为特定角色的缺省参数值。 如果使用了 `IN DATABASE` 选项，则仅仅指定的角色和数据库可以设置或者清除相关参数。

特定角色的参数设置仅仅在登录时起效；SET ROLE和SET SESSION AUTHORIZATION不能进行特定角色的参数设置。

参阅SET和Chapter 18获取有关允许的参数名称和数值的更多信息。

注意

使用CREATE ROLE增加新角色，使用DROP ROLE删除旧角色。

`ALTER ROLE` 不能改变角色的成员关系。可以使用GRANT和REVOKE做这个事情。

使用这个命令指定一个未加密的密码时必须小心，因为密码将以明文方式传送到服务器，并且可能被客户端命令历史记录或者被服务器日志记录。 psql包含一个可以用来安全修改角色密码的 `\password` 命令，这个命令不会暴露明文的密码。

也可以把会话缺省参数值与数据库绑定而不是与角色绑定；参阅ALTER DATABASE。 如果有冲突，那么指定角色加数据库的参数设置将覆盖指定角色的参数设置，而后者又可以覆盖指定数据库的参数设置。

例子

改变一个角色的口令：

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

清除一个角色的口令:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

改变口令失效的日期，声明口令应该在2015年5月4日中午失效，时区比UTC早一个小时：

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

设置一个口令永久有效：

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

授予一个角色创建其它角色和新数据库的权限：

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

给一个角色设置非缺省的maintenance_work_mem参数值：

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

给一个角色设置非缺省的、指定数据库的client_min_messages参数值：

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

兼容性

`ALTER ROLE` 语句是一个PostgreSQL扩展。

参见

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER RULE

Name

ALTER RULE -- 修改一个规则的定义

Synopsis

```
ALTER RULE _name_ ON _table_name_ RENAME TO _new_name_
```

描述

`ALTER RULE`修改一个规则的定义。目前仅可以修改规则的名称。

要使用`ALTER RULE`，用户必须是应用了指定规则的表或视图的所有者。

参数

`_name_`

要修改规则的名称。

`_table_name_`

应用了指定规则的表或视图的名称（可以有模式修饰符）。

`_new_name_`

规则的新名称。

例子

将一个规则的名称进行重命名：

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

兼容性

`ALTER RULE`是PostgreSQL的扩展，这个语句会重写系统内容。

参见

CREATE RULE, DROP RULE

ALTER SCHEMA

Name

ALTER SCHEMA -- 修改一个模式的定义

Synopsis

```
ALTER SCHEMA _name_ RENAME TO _new_name_
ALTER SCHEMA _name_ OWNER TO _new_owner_
```

描述

`ALTER SCHEMA`修改一个模式的定义。

要使用`ALTER SCHEMA`，你必须是该模式的所有者。 要给一个模式重命名，你必须在该数据库上有`CREATE`权限。 要修改所有者，你还必须是新的所有角色的直接或间接成员， 并且该成员必须在此数据库上有`CREATE`权限。 （超级用户自动拥有全部权限。）

参数

`_name_`

现有模式的名字。

`_new_name_`

模式的新名字，新名字不能以`pg_`开头，因为这些名字是保留给系统模式的。

`_new_owner_`

模式的新所有者。

兼容性

SQL标准里没有`ALTER SCHEMA`语句。

参见

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

Name

ALTER SEQUENCE -- 更改一个序列生成器的定义

Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] _name_ [ INCREMENT [ BY ] _increment_ ]
    [ MINVALUE _minvalue_ | NO MINVALUE ] [ MAXVALUE _maxvalue_ | NO MAXVALUE ]
    [ START [ WITH ] _start_ ]
    [ RESTART [ [ WITH ] _restart_ ] ]
    [ CACHE _cache_ ] [ [ NO ] CYCLE ]
    [ OWNED BY { _table_name_._column_name_ | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] _name_ OWNER TO _new_owner_
ALTER SEQUENCE [ IF EXISTS ] _name_ RENAME TO _new_name_
ALTER SEQUENCE [ IF EXISTS ] _name_ SET SCHEMA _new_schema_
```

描述

`ALTER SEQUENCE` 命令修改一个现有的序列发生器的参数。 任何没有明确在 `ALTER SEQUENCE` 命令里声明的参数都将保留原先的设置。

要使用 `ALTER SEQUENCE` ，你必须是该序列的所有者。 要改变一个序列的模式，你必须在新的模式上有 `CREATE` 权限。 要改变一个序列的所有者，你必须也是新的所有角色的直接或者间接的成员，并且那个角色必须有序列模式上的 `CREATE` 权限。 （这些约束强制在改变所有者时只能做删除或者创建序列时能做的操作。然而，超级用户可以以任何方式改变任意序列的成员关系。）

参数

`_name_`

一个要修改的序列的名字(可以有模式修饰)。

`IF EXISTS`

当序列不存在时使用该选项不会出现错误信息，仅有一个提示信息。

`_increment_`

`INCREMENT BY` `_increment_` 选项是可选的。 一个正数会让序列成为递增序列，负数则成为递减序列。如果没有声明，将沿用原来的递增值。

minvalue``NO MINVALUE

`MINVALUE` `_minvalue_`是一个可选选项，它决定一个序列可以生成的最小的值。 如果声明了`NO MINVALUE`，那么将使用缺省值，对于递增和递减的序列分别是1和-2⁶³-1。 如果没有声明则沿用当前的最小值。

`_maxvalue_` `NO MAXVALUE`

`MAXVALUE` `_maxvalue_`是一个可选选项，它决定一个序列可以生成的最大的值。 如果声明了`NO MAXVALUE`那么将使用缺省值，对于递增和递减的序列分别是2⁶³-1和-1。 如果没有声明则沿用当前的最大值。

`_start_`

`START WITH` `_start_`是一个可选选项，它修改序列的起始值。 这对序列当前值没有影响；它仅设置将来的`ALTER SEQUENCE RESTART`命令将会使用的值。

`_restart_`

`RESTART [ WITH` `_restart_`]是一个可选选项，它改变序列的当前值。 这相当于用`is_called` = `false`参数调用`setval` 函数 ：指定的值将会通过`nextval`的*next*调用返回。 不使用`_restart_`参数值来使用`RESTART`选项相当于使用`CREATE SEQUENCE`或由`ALTER SEQUENCE START WITH`设置的初始值。

`_cache_`

`CACHE` `_cache_`选项打开序列号预分配功能以及为了快速存取而在内存中存储的功能。 最小值是1（表示每次只能生成一个数值，没有缓冲）。如果没有声明，将沿用旧的缓冲值。

`CYCLE`

可选的 `CYCLE` 选项用于设置升序序列或是降序序列在达到`_maxvalue_`或者`_minvalue_`的时候循环使用。 如果达到了极限， 那么生成的下一个数字将分别是`_minvalue_`或者`_maxvalue_`。

`NO CYCLE`

如果声明了可选的`NO CYCLE`选项，任何在序列达到其极限后对`nextval`的调用都将返回错误。 如果未声明`CYCLE`或者`NO CYCLE`，那么将沿用原有的循环行为。

`OWNED BY` `_table_name_`.`_column_name_` `OWNED BY NONE`

`OWNED BY`选项将序列和一个表的特定字段关联， 这样，如果那个字段(或整个表)被删除了，那么序列也将被自动删除。 如果序列已经与表有关联后，使用这个选项后新关联将覆盖旧有的关联。 指定的表必须与序列的所有者相同并且在同一个模式中。 使用`OWNED BY NONE`将删除任何已经存在的关联，也就是让该序列变成"独立"序列。

`_new_owner_`

序列新所有者的用户名。

`_new_name_`

序列的新名称。

`_new_schema_`

序列的新模式。

注意

为避免从同一序列获取序列值的并发事务阻塞，在序列产生参数上的 `ALTER SEQUENCE` 的影响从不回滚；这些改变会立刻生效并且是不可逆的；然而，`OWNED BY`、`OWNER TO`、`RENAME TO` 和 `SET SCHEMA` 选项会引起可被回滚的普通更新。

除了当前的正在运行取值操作，`ALTER SEQUENCE` 不会立刻影响后台的 `nextval` 结果。序列在使用完所有缓存的序列值后才会使用变化后的序列参数值。当前后台正在运行取值操作将会立刻受到影响。

`ALTER SEQUENCE` 不影响序列的 `currval` 状态。（在PostgreSQL 8.3之前，有时会影响。）

由于历史原因，`ALTER TABLE` 也可用于序列；但是 `ALTER TABLE` 仅有的与序列有关的变化等价于前面所示的形式。

例子

将序列serial设置为从105重新开始取值：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

`ALTER SEQUENCE` 遵从SQL标准，但 `START WITH`、`OWNED BY`、`OWNER TO`、`RENAME TO` 和 `SET SCHEMA` 选项是 PostgreSQL的扩展。

参见

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

Name

ALTER SERVER -- 更改外部服务器的定义

Synopsis

```
ALTER SERVER _name_ [ VERSION '_new_version_' ]
    [ OPTIONS ( [ ADD | SET | DROP ] _option_ ['_value_'] [, ... ] ) ]
ALTER SERVER _name_ OWNER TO _new_owner_
ALTER SERVER _name_ RENAME TO _new_name_
```

描述

`ALTER SERVER` 更改外部服务器的定义。 第一种形式改变外部服务器版本字符串 或者服务器的通用选项（至少需要一个选项）。 第二种形式改变外部服务器的所有者。

要更改外部服务器，你必须是外部服务器的所有者。 此外要更改所有者，你必须是外部服务器的所有者并且也是新的所有者角色的直接或者间接成员，并且你必须对外部服务器的外部数据封装器有 `USAGE` 权限。 （注意超级用户能自动满足所有这些 条件。）

参数

`_name_`

已有服务器的名称。

`_new_version_`

新的服务器版本。

`OPTIONS ( [ ADD | SET | DROP ]` `_option_` [' `_value_` '] [, ...])

外部服务器更新选项。 `ADD` 、 `SET` 和 `DROP` 声明要被执行的动作。 若没有明确指定操作，则假定为 `ADD` 。 选项名称必须是唯一的;名称和数值也确认是使用服务器的外部数据封装器的库。

`_new_owner_`

外部服务器的新所有者的用户名。

`_new_name_`

外部服务器的新名称。

例子

更改服务器foo, 添加连接选项:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

更改服务器foo，更改版本, 更改host选项:

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

兼容性

`ALTER SERVER` 遵守ISO/IEC 9075-9 (SQL/MED)标准。 `OWNER TO` 和 `RENAME` 选项是PostgreSQL扩展。

参见

CREATE SERVER, DROP SERVER

ALTER TABLE

Name

ALTER TABLE -- 修改表的定义

Synopsis

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] _name_ [ * ]
    _action_ [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] _name_ [ * ]
    RENAME [ COLUMN ] _column_name_ TO _new_column_name_
ALTER TABLE [ IF EXISTS ] [ ONLY ] _name_ [ * ]
    RENAME CONSTRAINT _constraint_name_ TO _new_constraint_name_
ALTER TABLE [ IF EXISTS ] _name_
    RENAME TO _new_name_
ALTER TABLE [ IF EXISTS ] _name_
    SET SCHEMA _new_schema_

其中`_action_` 可以是以选项之一：

    ADD [ COLUMN ] _column_name_ _data_type_ [ COLLATE _collation_ ] [ _column_constraint
    DROP [ COLUMN ] [ IF EXISTS ] _column_name_ [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] _column_name_ [ SET DATA ] TYPE _data_type_ [ COLLATE _collation_ ]
    ALTER [ COLUMN ] _column_name_ SET DEFAULT _expression_
    ALTER [ COLUMN ] _column_name_ DROP DEFAULT
    ALTER [ COLUMN ] _column_name_ { SET | DROP } NOT NULL
    ALTER [ COLUMN ] _column_name_ SET STATISTICS _integer_
    ALTER [ COLUMN ] _column_name_ SET ( _attribute_option_ = _value_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ RESET ( _attribute_option_ [, ... ] )
    ALTER [ COLUMN ] _column_name_ SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD _table_constraint_ [ NOT VALID ]
    ADD _table_constraint_using_index_
    VALIDATE CONSTRAINT _constraint_name_
    DROP CONSTRAINT [ IF EXISTS ]  _constraint_name_ [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ _trigger_name_ | ALL | USER ]
    ENABLE TRIGGER [ _trigger_name_ | ALL | USER ]
    ENABLE REPLICA TRIGGER _trigger_name_
    ENABLE ALWAYS TRIGGER _trigger_name_
    DISABLE RULE _rewrite_rule_name_
    ENABLE RULE _rewrite_rule_name_
    ENABLE REPLICA RULE _rewrite_rule_name_
    ENABLE ALWAYS RULE _rewrite_rule_name_
    CLUSTER ON _index_name_
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET ( _storage_parameter_ = _value_ [, ... ] )
    RESET ( _storage_parameter_ [, ... ] )
    INHERIT _parent_table_
    NO INHERIT _parent_table_
    OF _type_name_
    NOT OF
    OWNER TO _new_owner_
    SET TABLESPACE _new_tablespace_

and `_table_constraint_using_index_` is:

    [ CONSTRAINT _constraint_name_ ]
    { UNIQUE | PRIMARY KEY } USING INDEX _index_name_
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

描述

`ALTER TABLE` 改变一个现存表的定义。它有好几种子形式：

`ADD COLUMN`

这种形式使用和CREATE TABLE一样的语法向表中增加一个新的字段。

`DROP COLUMN [ IF EXISTS ]`

这种形式从表中删除一个字段。和这个字段相关的索引和表约束也会被自动删除。 如果任何表之外的对象依赖于这个字段，必须使用 `CASCADE` 选项，比如外键约束、 视图等。 如果使用了 `IF EXISTS` 选项并且相关字段不存在，则在删除时不会显示错误，仅会有一个提示信息。

`IF EXISTS`

如果使用了 `IF EXISTS` 选项并且表不存在，则在删除时不会显示错误，仅会有一个提示信息。。

`SET DATA TYPE`

这种形式改变表中一个字段的类型。该字段涉及的索引和简单的表约束将被自动地转换为使用新的字段类型，方法是重新分析最初提供的表达式。 可选的 `COLLATE` 选项定义了新列的字符集排序方式，如果不加这个选项，则排序方式使用新类型的缺省值。 可选的 `USING` 选项定义如何从旧的字段值里计算新的字段值；如果省略，那么缺省的转换就是从旧类型向新类型的赋值转换。 如果从旧数据类型到新类型没有隐含或者赋值的转换，那么必须提供一个 `USING` 选项。

`SET` / `DROP DEFAULT`

这种形式设置或是删除一列的缺省值。缺省值仅会对后续的 `INSERT` 或是 `UPDATE` 命令，不会影响已经在表中存在的记录。

`SET` / `DROP NOT NULL`

这种形式修改一个字段是否允许 NULL 值或者拒绝 NULL 值。如果表中字段里包含非 NULL 值 ，那么你只可以使用 `SET NOT NULL` 选项。

`SET STATISTICS`

这种形式为后续的ANALYZE操作设置每列的数据统计目标值。 目标值可以设置为0至10000；相应地， 设置为-1则会反向使用系统缺省的统计目标值 (default_statistics_target)。 有关PostgreSQL查询规划器使用数据统计方面的更多信息，可以参阅Section 14.2。

`SET (` `_attribute_option_` = `_value_` [, ...]) `RESET (` `_attribute_option_` [, ...])

这种形式设置或者重置每个属性的参数值。目前，已定义的属性参数值是 `n_distinct` 和 `n_distinct_inherited` ， 它们会覆盖由随后的ANALYZE操作所估算的number-of-distinct-values。 `n_distinct` 影响表本身的统计值，而 `n_distinct_inherited` 影响表及其继承子表的统计。 当设置为一个正值时， `ANALYZE` 将会假定列准确包含明确的非空值的指定数目。 当设置为一个必须大于或者等于-1的负值时， `ANALYZE` 将会假定在列中的不同的非空值的数目与表的大小关系是线性的； 确切的统计通过将估算的表大小与给定数字的绝对值相乘来统计。 例如， 值-1意味着在此列中的所有值是不同的，值-0.5意味着每个值平均出现两次。 当表的大小随时间变化时这是很有效的，尽管表中行数的乘法运算在查询规划计时前是不会这样计算的， 声明一个0值来正常地恢复到估计不同数值的数目。 要获取关于使用PostgreSQL查询优化器做统计的信息，请参阅Section 14.2。

`SET STORAGE`

这种形式为一个字段设置存储模式。这个设置控制这个字段是内联保存还是保存在一个TOAST附属的表里，以及数据是否要压缩。 `PLAIN` 选项必须用于定长的数值，比如 `integer` 并且是内联的、不压缩的。 `MAIN` 用于内联、可压缩的数据。 `EXTERNAL` 用于外部保存、不压缩的数据， `EXTENDED` 用于外部的压缩数据。 `EXTENDED` 是大多数支持非 `PLAIN` 存储的数据的缺省值。 使用 `EXTERNAL` 将会在 `text` 和 `bytea` 字段上的字符串操作更快，但付出的代价是增加了存储空间。 请注意 `SET STORAGE` 本身并不改变表上的任何东西，只是设置将来的表操作时，建议使用的策略。参阅Section 58.2获取更多信息。

`ADD` `_table_constraint_` [NOT VALID]

这种形式给表增加一个新的约束，使用的语法和CREATE TABLE一样，而与 `NOT VALID` 选项组合时，这种约束仅在对外键和CHECK类药束有效。 如果约束条件增加了 `NOT VALID` 选项后，表中已有记录是否满足初始约束检查会被跳过。这种约束会对后续的新增记录或是记录更新产生影响（在外键约束的情况下， 除非是参照的表中有匹配的记录，否则这样的情形会出错；而在CHECK约束下，除非是新增记录满足CHECK约束条件，否则也会也错）。 但数据库自身不会认定这种约束会对表中所有记录生效，除非是在使用了 `VALIDATE CONSTRAINT` 选项进行了有效性检查。

`ADD` `_table_constraint_using_index_`

这种形式根据已有的唯一索引给表增加新的 `PRIMARY KEY` 或 `UNIQUE` 约束。索引中所有的列也会在包含在约束中。

索引中不能含有表达式列或者是局部索引。另外索引的缺省排序方式应是B-tree类型。这些限制保证了索引会与通过正常 `ADD PRIMARY KEY` 或 `ADD UNIQUE` 选项生成的索引相同。

如果使用了 `PRIMARY KEY` 选项，则索引的列不能已定义为 `NOT NULL` ，因为这个选项会对涉及的列去执行 `ALTER COLUMN SET NOT NULL` 。 这也会对全表数据进行扫描以验证该列是否包含空值， （全表扫描很慢）， 在其他情况下，这是一个很快的操作。

如果指定了一个约束名，索引将会被重命名为指定的约束中的名称。相反，约束会被命名为索引名。

在这个命令执行后，约束就相当于是索引的"所有者"了，就如同使用了 `ADD PRIMARY KEY` 或 `ADD UNIQUE` 命令创建的索引一样。 特别要注意的事，这种情况下删除约束也会清除了索引。

> **Note:** 在新增约束时使用已有的索引对新约束增加时对表记录的较长时间不能更新的问题有较好的帮助。 使用 `CREATE INDEX CONCURRENTLY` 指令可以实现这种方式，参见下面的示例。

`VALIDATE CONSTRAINT`

这种形式用于验证一个外键或是一个使用 `NOT VALID` 选项创建的检查类约束，通过扫描全表来保证所有记录都符合约束条件。 如果约束已标记为有效时，什么操作也不会发生。

对大表的记录进行验证一般是一个很长的过程，并且目前这种操作还需要 `ACCESS EXCLUSIVE` 排他类锁。 初始的不同验证可以将验证工作后延至系统不忙时进行，或者通过先花一点额外的时间来纠正可能存在的错误以防止新错误的发生。

`DROP CONSTRAINT [ IF EXISTS ]`

这种形式删除一个表中指定的约束，如果使用了 `IF EXISTS` 选项并且约束并不存在时，也不会有错误产生，仅会有一个提示信息。

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ] TRIGGER`

这种形式禁用或者启用属于该表的触发器。一个被关闭掉的触发器是系统仍然知道的，但是在触发器事件发生的时候不会被执行。 对于一个推迟了的触发器，在触发事件发生的时候会检查打开状态，触发器相关的函数实际也不会执行。 可以通过指定名字的方法启用或者禁用任意一个触发器，或者是该表上的所有触发器，或者只是用户自定义的触发器(这个选项排除了那些用于实现外键约束或是可延迟的唯一性约束或是排他性约束的触发器)。 启用或者禁用约束触发器要求超级用户权限；这么做的时候应该小心，因为如果触发器不执行的话，约束保证的数据完整性也就没有办法确保了。 触发器启动原理也受配置变量session_replication_role影响。 简单启动的触发器将会在复制任务为"初始"(默认情况)或者"本地"时启动。 配置为 `ENABLE REPLICA` 的触发器将会仅在会话为"replica" 模式时启动，而配置为 `ENABLE ALWAYS` 的触发器将总是会启动，无论是否为当前复制模式。

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ] RULE`

这种形式配置属于表的重写规则制定。一个已禁用的规则对系统来说仍然是可知的，但在查询重写期间是不被应用的。 语义为关闭/启动触发器。这个配置对 `ON SELECT` 规则来说是可忽略的，常常用来保持视图工作，即使当前会话处于一个非默认的复制角色中。

`CLUSTER ON`

这种形式为将来的CLUSTER操作选择默认索引。 实际上并没有重新盘簇化处理该表。

`SET WITHOUT CLUSTER`

这种形式从表中删除最近一次用到的CLUSTER索引定义。这会影响将来不声明索引的盘簇化表的操作。

`SET WITH OIDS`

这种形式向表中增加一个 `oid` 系统字段（参见Section 5.4）。 如果表中已存在有OID字段，则操作对表无任何影响。

注意这种形式与 `ADD COLUMN oid oid` 选项并不相同，后者对给表增加一个普通的字段，只不过它的名称恰好是叫 `oid`，并非是系统字段。

`SET WITHOUT OIDS`

这种形式从表中删除 `oid` 系统字段。它和 `DROP COLUMN oid RESTRICT` 完全相同，只不过是如果表上已经没有 `oid` 字段的时候不会报错。

`SET (` `_storage_parameter_` = `_value_` [, ...])

这种形式改变表的一个或者更多存储参数。参阅存储参数获取关于可用参数的详细信息。 请注意表的内容将不会因为此命令被立刻调整；根据此参数你可能需要重写此表来得到希望的效果。 这可以通过VACUUM FULL、CLUSTER或者 `ALTER TABLE` 命令中的选项之一来实现。

> **Note:** 尽管 `CREATE TABLE` 允许 `OIDS` 在 `WITH (``_storage_parameter_` )语义中声明，但 `ALTER TABLE` 不会将 `OIDS` 作为一个存储参数。 相反地，要使用 `SET WITH OIDS` 和 `SET WITHOUT OIDS` 形式来更改OID状态。

`RESET (` `_storage_parameter_` [, ...])

这种形式重置表的一个或多个存储参数为缺省值。与 `SET` 选项一样，根据参数的不同可能需要重写表才能获得想要的效果。

`INHERIT` `_parent_table_`

这种形式将目标表添加为指定父表的新子表。 之后在父表上的查询将包含目标表中的记录。 要被添加为一个子表，目标表必须已经包含所有与父表相同的字段(除此之外 当然也可以包含一些其它字段)，这些字段的数据类型必须匹配，并且如果父表的字段有 `NOT NULL` 约束的话子表的相应字段也必须有 `NOT NULL` 约束。

所有父表的 `CHECK` 约束必须同时与子表的约束匹配。 不过一些标记为不可继承类的约束（类似使用 `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT` 创建的约束）不包括在内， 所有子表匹配的约束也不能标记为不可继承。 目前 `UNIQUE` 、 `PRIMARY KEY` 和 `FOREIGN KEY` 约束不被考虑在内，但是将来可能会有所改变。

`NO INHERIT` `_parent_table_`

这种形式从指定父表的子表列表中删除目标表。这样，在父表上的查询将不再目标表中的记录。

`OF` `_type_name_`

这种形式将表链接至一种复合类型，就好象是使用 `CREATE TABLE OF` 选项创建表一样。 表的字段的名称和类型必须精确匹配复合类型中的定义，不过 `oid` 系统字段允许不一样。 表不能是从任何其他表继承的。 这些限制确保 `CREATE TABLE OF` 选项允许一个相同的表定义。

`NOT OF`

这种形式将一个与某类型进行关联的表进行关联的解除。

`OWNER`

这种形式改变表、序列或是视图的所有者为一个指定的用户。

`SET TABLESPACE`

这种形式更改表的表空间为一个指定的表空间，并将与这个表相关的数据文件移至新的表空间。 表上如果有索引，一般不会移动。不过它们也可以通过使用 `SET TABLESPACE` 命令单独移动，参见CREATE TABLESPACE。

`RENAME`

`RENAME` 形式改变一个表(或者索引、序列、视图)的名称，表中单个字段的名称，或是表中约束的名称。 它们对存储的数据没有影响。

`SET SCHEMA`

这种形式把表移动到另外一个模式。相关的索引、约束、序列都跟着移动。

除了 `RENAME` 和 `SET SCHEMA` 之外所有动作都可以组合在一个多次修改列表中同时使用。 比如，可以在一个命令里增加几个字段和/或修改几个字段的类型。 对于大表，这么做特别有用，因为只需要对该表做一次处理。

要使用 `ALTER TABLE` ，你必须是该表的所有者。 要修改一个表的模式，你还必须在新模式上拥有 `CREATE` 权限。 要把该表添加为一个父表的新子表，你必须同时是父表的所有者。 要修改所有者，你还必须是新的所有角色的直接或间接成员，并且该成员必须在此表的模式上有 `CREATE` 权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建表不能做的事情。不过，超级用户可以以任何方式修改任意表的所有权。） 增加一个字段或是改变字段的类型或是使用 `OF` 选项，你也必须对那个数据类型有 `USAGE` 权限。

参数

`_name_`

要修改的已有表的名称（可以有模式修饰）。若声明了 `ONLY` 选项，则只有那个表被更改。若未声明 `ONLY` ，该表及其所有子表都将会被更改。 另外，可以在表名称后面精确地增加 `*` 选项来指定包括子表。

`_column_name_`

现存或新的字段名称。

`_new_column_name_`

现存字段的新名称。

`_new_name_`

表的新名称。

`_type_`

新字段的类型，或者现存字段的新类型。

`_table_constraint_`

新的表约束。

`_constraint_name_`

要删除的现有约束的名字。

`CASCADE`

级联删除依赖于被依赖字段或者约束的对象(比如引用该字段的视图)。

`RESTRICT`

如果字段或者约束还有任何依赖的对象，则拒绝删除该字段。这是缺省行为。。

`_trigger_name_`

要启用或者禁用的单个触发器的名字。

`ALL`

启用或者禁用所有属于该表的触发器。（如果任何触发器属于内部会产生约束的触发器，这要求超级用户权限，例如那些用于执行外键约束或者可推迟的独特性和排除约束。）

`USER`

启用或者禁用所有属于表的触发器，除了任何属于内部会产生约束的触发器，例如那些用于执行外键约束或者可推迟的独特性和排除约束。）

`_index_name_`

要标记为盘簇化的表上面的索引名字。

`_storage_parameter_`

表的存储参数的名字。

`_value_`

表的存储参数的新值，根据参数的不同，可能是一个数字或单词。

`_parent_table_`

将要与该表建立/取消关联的父表。

`_new_owner_`

该表的新所有者的用户名。

`_new_tablespace_`

这个表将要移动到的表空间名字。

`_new_schema_`

表将移动到的新模式的名字。

注意

`COLUMN` 关键字是多余的，可以省略。。

如果用 `ADD COLUMN` 增加一个字段，那么所有表中现有行都初始化为该字段的缺省值(如果没有声明 `DEFAULT` 子句，那么就是 NULL)。

添加一个非空缺省值列或者改变一个原有列的类型需要重写整个表和索引。 不过也有例外，如果使用 `USING` 选项不改变字段的内容并且字段的新旧类型是二进制兼容的，也可以不重写表。 添加或者删除一个系统 `oid` 列同样需要重写整个表。 大表及索引的重写可能需要非常长的时间，并且也临时需要两倍的磁盘空间。

增加一个 `CHECK` 或 `NOT NULL` 约束将会扫描该表以保证现有的行符合约束要求。

提供在一个 `ALTER TABLE` 里面声明多个修改的主要原因是原先需要的对表的多次扫描和重写可以组合成一个操作。

`DROP COLUMN` 命令并不是物理上把字段删除，而只是简单地 把它标记为对 SQL 操作不可见。 随后对该表的插入和更新将在该字段存储一个 NULL 。 因此，删除一个字段是很快的，但是它不会立即释放表在磁盘上的空间，因为被删除 了的字段占据的空间还没有回收。 这些空间将随着现有的行的更新而得到回收。 (在删除系统 `oid` 列时，方式有点不同，这个操作是直接执行了一次表的重写。)

要强制立刻执行一次表的重写，可以使用VACUUM FULL、CLUSTER或是ALTER TABLE命令中的一些形式。 这些命令从可见的语义上不会对表产生更新，但会清除不再有用的数据。

使用 `SET DATA TYPE` 命令中的 `USING` 选项实际上可以指定任何表达式涉及到记录旧值； 也就是说，它可以引用除了正在被转换的字段之外其它的字段。 这样，就可以用 `SET DATA TYPE` E语法做非常普遍性的转换。 因为这个灵活性， `USING` 表达式并没有应用于该字段的缺省值(如果有的话)；结果可能不是缺省表达式要求的常量表达式。 这就意味着如果从旧类型到新类型没有隐含或者赋值 转换的话，那么即使存在 `USING` 选项， `SET DATA TYPE` 也可能无法把缺省值转换成新的类型。 在这种情况下，应该用 `DROP DEFAULT` 先删除缺省值，执行 `ALTER TYPE` ，然后使用 `SET DEFAULT` 增加一个 合适的新缺省值。 类似的考虑也适用于涉及该字段的索引和约束。

如果一个表有子表，那么如果不在子表上做同样的修改的话，就不允许在父表上增加、 重命名、修改一个字段的类型。 也就是说， `ALTER TABLE ONLY` 将被拒绝执行。 这样就保证了子表总是有和父表匹配的字段。

一个递归 `DROP COLUMN` 操作将只有在子表并不从任何其它父表中继承该字段并且从来没有独立定义该字段的时候才能删除一个子表的字段。 一个非递归的 `DROP COLUMN` (也就是 `ALTER TABLE ONLY ... DROP COLUMN`) 从来不会删除任何子表字段， 而是把他们标记为独立定义的(而不是继承的)。 。

`TRIGGER`、`CLUSTER`、`OWNER`和`TABLESPACE`的操作绝不会递归影响到子表；也就是说，它们的行为就像总是声明了`ONLY`一样。只有没有标记为`NO INHERIT`的`CHECK`约束才能添加一个递归性的约束。

不允许更改系统表结构的任何部分。

请参考CREATE TABLE部分获取更多有效参数的描述。Chapter 5章节里有更多有关继承的信息。。

例子

向表中增加一个`varchar`字段：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

从表中删除一个字段：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

在一个操作中修改两个现有字段的类型：

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

使用一个`USING`选项，把一个包含 UNIX 时间戳的 integer 字段转化成`timestamp with time zone`字段：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

同样地，当字段有一个不会自动转换成新类型的缺省值表达式时：

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

对现有字段改名：

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

更改现有表的名字：

```
ALTER TABLE distributors RENAME TO suppliers;
```

更改现有约束的名字：

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

给一个字段增加一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

从一个字段里删除一个非空约束：

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

给一个表增加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

给一个表且不包含其子表增加一个检查约束：

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERI
```

（这个检查约束也不会被以后新增的子表继承。）

删除一个表及其所有子表的检查约束：

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

从表中删除一个检查约束（不包含子表）：

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（这个检查约束对所有子表仍保留。）

向表中增加一个外键约束：

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses
```

给表增加一个(多字段)唯一约束：

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

给一个表增加一个自动命名的主键约束，要注意的是一个表只能有一个主键：

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

把表移动到另外一个表空间：

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

把表移动到另外一个模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

重新创建一个主键约束，并且在主键索引在创建时不影响记录的更新：

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

兼容性

`ADD` (不包含`USING INDEX`)、`DROP`、`SET DEFAULT`和`SET DATA TYPE`(不包含`USING`)形式与 SQL 标准兼容。 其它形式是PostgreSQL对 SQL 标准 的扩展。还有，在一个`ALTER TABLE`命令里声明多个操作也是扩展。

`ALTER TABLE DROP COLUMN`可以用于删除表中的唯一的一个字段，留下一个零字段的表。这是对 SQL 的扩展，它不允许零字段表。

参见

CREATE TABLE

ALTER TABLESPACE

Name

ALTER TABLESPACE -- 修改一个表空间的定义

Synopsis

```
ALTER TABLESPACE _name_ RENAME TO _new_name_
ALTER TABLESPACE _name_ OWNER TO _new_owner_
ALTER TABLESPACE _name_ SET ( _tablespace_option_ = _value_ [, ... ] )
ALTER TABLESPACE _name_ RESET ( _tablespace_option_ [, ... ] )
```

描述

`ALTER TABLESPACE` 改变一个表空间的定义。

要使用 `ALTER TABLESPACE`， 你必须是该表空间的所有者。 要修改所有者，你还必须是新的所有角色的直接或间接成员。不过，超级用户自动获得这些权限。

参数

`_name_`

一个现有的表空间的名称。

`_new_name_`

表空间的新名字。新名字不能以 `pg_` 开头，因为这样的名字保留给系统表空间使用了。

`_new_owner_`

表空间的新所有者。

`_tablespace_option_`

可以设置或者重置的表空间参数。目前，可以设置的参数是 `seq_page_cost` 和 `random_page_cost` 。 为一个表空间设置任一个值，将会覆盖在那个表空间中从表中阅读页的成本优化器的一般估计值，正如同通过同名的配置参数建立的。 （参阅seq_page_cost，random_page_cost）。 如果表空间所在的磁盘是一个更快或更慢的其它io子系统，那么这些参数是有用的。

例子

把表空间 `index_space` 重命名为 `fast_raid`：

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

改变表空间 `index_space` 的所有者：

```
ALTER TABLESPACE index_space OWNER TO mary;
```

兼容性

SQL标准里没有 `ALTER TABLESPACE` 语句。

参见

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

Name

ALTER TEXT SEARCH CONFIGURATION -- 更改文本搜索配置的定义

Synopsis

```
ALTER TEXT SEARCH CONFIGURATION _name_
ADD MAPPING FOR _token_type_ [, ... ] WITH _dictionary_name_ [, ... ]
ALTER TEXT SEARCH CONFIGURATION _name_
ALTER MAPPING FOR _token_type_ [, ... ] WITH _dictionary_name_ [, ... ]
ALTER TEXT SEARCH CONFIGURATION _name_
ALTER MAPPING REPLACE _old_dictionary_ WITH _new_dictionary_
ALTER TEXT SEARCH CONFIGURATION _name_
ALTER MAPPING FOR _token_type_ [, ... ] REPLACE _old_dictionary_ WITH _new_dictionary_
ALTER TEXT SEARCH CONFIGURATION _name_
DROP MAPPING [ IF EXISTS ] FOR _token_type_ [, ... ]
ALTER TEXT SEARCH CONFIGURATION _name_ RENAME TO _new_name_
ALTER TEXT SEARCH CONFIGURATION _name_ OWNER TO _new_owner_
ALTER TEXT SEARCH CONFIGURATION _name_ SET SCHEMA _new_schema_
```

描述

`ALTER TEXT SEARCH CONFIGURATION`更改文本搜索配置的定义。 您可以将映射从字串类型调整为字典，或者改变配置的名称或者所有者。

要使用`ALTER TEXT SEARCH CONFIGURATION`，您必须是配置的所有者。

参数

`_name_`

已有文本搜索配置的名称（可以有模式修饰）。

`_token_type_`

与配置的语法解析器关联的字串类型的名称。

`_dictionary_name_`

对指定的字串类型将要被搜索使用的文本搜索字典的名称。 如果有多个字典，则它们会按指定的顺序搜索。

`_old_dictionary_`

映身中拟被替换的文本搜索字典名称。

`_new_dictionary_`

替换 `_old_dictionary_` 的文本搜索字典的名称。

`_new_name_`

文本搜索配置的新名称。

`_new_owner_`

文本搜索配置的新所有者。

`_new_schema_`

文本搜索配置的新模式名。

`ADD MAPPING FOR` 选项安装一个被特定字串类型搜索用的字典列表； 如果已经有任何一个字串类型的映射，系统将会报错。 `ALTER MAPPING FOR` 选项也有相同功能，但是它会首先清除已有的字串类型的映射。 `ALTER MAPPING REPLACE` 选项使用 `_new_dictionary_` 替换 `_old_dictionary_` ，只要后者一出现便会被替换。 这些仅仅在 `FOR` 选项出现时对特定字串类型来操作，或者在不出现时对所有配置映射来操作。 `DROP MAPPING` 选项会删除所有特定字串类型相关的字典，导致那些类型的字串被文本搜索配置忽略。 除非是使用了 `IF EXISTS` 选项，否则如果对指定的字串类型无映射，则是错误的。

例子

以下示例执行后，在my_config这个配置方案内使用之内使用 `english` 字典时，会用 `swedish` 字典代替 `english` 字典。

```
ALTER TEXT SEARCH CONFIGURATION my_config
ALTER MAPPING REPLACE english WITH swedish;
```

兼容性

在SQL标准中没有 `ALTER TEXT SEARCH CONFIGURATION` 语句。

参见

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

Name

ALTER TEXT SEARCH DICTIONARY -- 更改文本搜索字典的定义。

Synopsis

```
ALTER TEXT SEARCH DICTIONARY _name_ (
_option_ [ = _value_ ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY _name_ RENAME TO _new_name_
ALTER TEXT SEARCH DICTIONARY _name_ OWNER TO _new_owner_
ALTER TEXT SEARCH DICTIONARY _name_ SET SCHEMA _new_schema_
```

描述

`ALTER TEXT SEARCH DICTIONARY` 更改文本搜索字典的定义。你可以修改字典的指定模板的配置参数，也可以改变字典的名称或者所有者。

要使用 `ALTER TEXT SEARCH DICTIONARY`，您必须是字典的所有者。

参数

`_name_`

已有文本搜索字典的名称（可以有模式修饰）。

`_option_`

为该字典设置的指定模板选项的名称。

`_value_`

用于指定模板选项的新值。如果等号和数值省略，则字典中该选项以前设置的数值也会被清除，此时这些参数会变为缺省值。

`_new_name_`

文本搜索字典的新名称。

`_new_owner_`

文本搜索字典的新所有者。

`_new_schema_`

文本搜索字典的新模式。

指定模板的选项可以以任何顺序显示。

例子

以下示例命令为Snowball-based字典改变省略词列表。其他参数保持不变。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

以下示例命令改变语言选项为荷兰，并删除完整的省略词选项。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

接下来的示例命令"updates"字典的定义，实际上没有做任何改变。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

（这可以运行的原因是，选项删除代码不会报错说是不存在这样的选项参数。） 在为字典修改配置文件时这个技巧是很有效的： `ALTER` 将会强制现有的数据库会话来重读配置文件，否则如果预先已经读过，将不会执行此操作。

兼容性

在SQL标准中没有 `ALTER TEXT SEARCH DICTIONARY` 语句。

参见

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

Name

ALTER TEXT SEARCH PARSER -- 更改一个文本搜索解析器的定义

Synopsis

```
ALTER TEXT SEARCH PARSER _name_ RENAME TO _new_name_
ALTER TEXT SEARCH PARSER _name_ SET SCHEMA _new_schema_
```

描述

`ALTER TEXT SEARCH PARSER` 更改一个文本搜索解析器的定义。目前，唯一支持的功能就是改变解析器的名称。

要使用 `ALTER TEXT SEARCH PARSER` ，您必须是超级用户。

参数

`_name_`

已有文本搜索解析器的名称（可有模式修饰）。

`_new_name_`

文本搜索解析器的新名称。

`_new_schema_`

文本搜索解析器的新模式名。

兼容性

在SQL标准中没有 `ALTER TEXT SEARCH PARSER` 语句。

参见

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

Name

ALTER TEXT SEARCH TEMPLATE -- 更改文本搜索模板的定义

Synopsis

```
ALTER TEXT SEARCH TEMPLATE _name_ RENAME TO _new_name_
ALTER TEXT SEARCH TEMPLATE _name_ SET SCHEMA _new_schema_
```

描述

`ALTER TEXT SEARCH TEMPLATE` 更改文本搜索模板的定义。目前，仅支持修改模板的名称。

要使用 `ALTER TEXT SEARCH TEMPLATE`，您必须是超级用户。

参数

`_name_`

已有文本搜索模板的名称（可有模式修饰）。

`_new_name_`

文本搜索模板的新名称。

`_new_schema_`

文本搜索模板的新模式名。

兼容性

在SQL标准中没有 `ALTER TEXT SEARCH TEMPLATE` 语句。

参见

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

Name

ALTER TRIGGER -- 修改一个触发器的定义

Synopsis

```
ALTER TRIGGER _name_ ON _table_name_ RENAME TO _new_name_
```

描述

`ALTER TRIGGER` 改变一个现有触发器的定义。 `RENAME` 选项修改触发器的名称，而不用改变触发器的定义。

你必须是该触发器作用的表的所有者才能改变其属性。

参数

`_name_`

需要修改的现有触发器的名称。

`_table_name_`

该触发器作用的表的名字。

`_new_name_`

现有触发器的新名字。

注意

临时打开或者关闭触发器的能力是由ALTER TABLE而不是 `ALTER TRIGGER` 提供的， 因为 `ALTER TRIGGER` 没有一次打开或者关闭所有表的触发器的选项。

例子

重新命名一个现有触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

兼容性

`ALTER TRIGGER` 是PostgreSQL对SQL标准的扩展。

参见

ALTER TABLE

ALTER TYPE

Name

ALTER TYPE -- 修改一个类型的定义。

Synopsis

```
ALTER TYPE _name_ _action_ [, ... ]
ALTER TYPE _name_ OWNER TO _new_owner_
ALTER TYPE _name_ RENAME ATTRIBUTE _attribute_name_ TO _new_attribute_name_ [ CASCADE | R
ALTER TYPE _name_ RENAME TO _new_name_
ALTER TYPE _name_ SET SCHEMA _new_schema_
ALTER TYPE _name_ ADD VALUE [ IF NOT EXISTS ] _new_enum_value_ [ { BEFORE | AFTER } _exis

`_action_`可以是下列选项之一：

    ADD ATTRIBUTE _attribute_name_ _data_type_ [ COLLATE _collation_ ] [ CASCADE | RESTRI
    DROP ATTRIBUTE [ IF EXISTS ] _attribute_name_ [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE _attribute_name_ [ SET DATA ] TYPE _data_type_ [ COLLATE _collation_
```

描述

`ALTER TYPE` 改变一个现有类型的定义。 有以下几种形式：

`ADD ATTRIBUTE`

这种形式给复合类型增加新的属性，与CREATE TYPE命令语法相同。

`DROP ATTRIBUTE [ IF EXISTS ]`

这种形式从复合类型删除一个属性。如果指定了 `IF EXISTS` 选项，并且属性不存在时，不会有错误产生。在这种情况下系统仅会有一个提示信息。

`SET DATA TYPE`

这种形式改变一种复合类型中某个属性的类型。

`OWNER`

这种形式改变类型的所有者。

`RENAME`

这种形式改变类型的名称或是一个复合类型中的一个属性的名称。

`SET SCHEMA`

这种形式将类型移至一个新的模式中。

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

这种形式给枚举类型增加一个新值。新值在枚举类型中的位置可通过对一个已有数值使用 `BEFORE` 或 `AFTER` 选项来指定。否则，新值会加在值列表的最后面。

如果使用了 `IF NOT EXISTS` 选项，即使枚举中已含有新值也不会报错，仅有提示信息。相反，新增一个枚举中已有的新值会产生错误提示。

`CASCADE`

自动级联更新需更新类型以及相关联的记录和继承它们的子表。

`RESTRICT`

如果需联动更新类型是已更新类型的关联记录，则拒绝更新。这是缺省选项。

`ADD ATTRIBUTE`、 `DROP ATTRIBUTE` 和 `ALTER ATTRIBUTE` 选项可以组合成一个列表同时处理。 例如，在一条命令中同时增加几个属性或是更改几个属性的类型是可以实现的。

要使用 `ALTER TYPE`，你必须是该类型的所有者。 要修改一个类型的模式，你还必须在新模式上拥有 `CREATE` 权限。 要修改所有者，你还必须是新的所有角色的直接或间接成员， 并且该成员必须在此类型的模式上有 `CREATE` 权限。 （这些限制强制了修改该所有者不会做任何通过删除和重建类型不能做的事情。 不过，超级用户可以以任何方式修改任意类型的所有权。） 要增加一个属性或是修改一个属性的类型，你也必须有该类型的 `USAGE` 权限。

参数

`_name_`

一个需要修改的现有的类型的名字(可以有模式修饰)。

`_new_name_`

该类型的新名称。

`_new_owner_`

新所有者的用户名 。

`_new_schema_`

该类型的新模式 。

`_attribute_name_`

拟增加、更改或删除的属性的名称。

`_new_attribute_name_`

拟改名的属性的新名称。

`_data_type_`

拟新增属性的数据类型，或是拟更改的属性的新类型名。

`_new_enum_value_`

拟加入枚举类型值列表中的新值。和所有枚举参数一样，它必须使用引号引起来。

`_existing_enum_value_`

用于设定将要新增的枚举值在枚举中前后位置的已存在的枚举参考值。和所有枚举参数一样，它必须使用引号引起来。

注意

`ALTER TYPE ... ADD VALUE` （这种形式给枚举增加一个新值）命令是不能在一个事务处理块内部执行的。

与处理枚举原有的列表值相比，新增一个枚举值有时有点慢，这一般出现在新增的枚举值使用了 `BEFORE` 或 `AFTER` 选项来 指定新值的位置，而不是使用缺省的选项放在列表最后。不过有时使用缺省放在最后也会慢（这在该枚举创建后其OID计数出现了"复位"情况）。 当然，这种性能上的损失是很小的。如果用户介意的话，可以通过删除这个枚举然后重新创建，或是导出数据库再导入即可恢复正常性能。

例子

重命名数据类型:

```
ALTER TYPE electronic_mail RENAME TO email;
```

要改变一个用户定义类型 `email` 的所有者为 `joe` :

```
ALTER TYPE email OWNER TO joe;
```

把用户定义类型 `email` 的模式改变为 `customers` :

```
ALTER TYPE email SET SCHEMA customers;
```

给一个数据类型增加一个新的属性：

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

给一个枚举类型增加一个指定位置的新值：

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

兼容性

增加或删除属性是SQL标准，其他选项都是PostgreSQL的扩展。

参见

CREATE TYPE, DROP TYPE

ALTER USER

Name

ALTER USER -- 修改一个数据库角色。

Synopsis

```
ALTER USER _name_ [ [ WITH ] _option_ [ ... ] ]

where `_option_` can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT _connlimit_
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD '_password_'
    | VALID UNTIL '_timestamp_'

ALTER USER _name_ RENAME TO _new_name_

ALTER USER _name_ SET _configuration_parameter_ { TO | = } { _value_ | DEFAULT }
ALTER USER _name_ SET _configuration_parameter_ FROM CURRENT
ALTER USER _name_ RESET _configuration_parameter_
ALTER USER _name_ RESET ALL
```

描述

`ALTER USER` 现在是ALTER ROLE命令的别名（即两者相同）。

兼容性

`ALTER USER` 是PostgreSQL的扩展，SQL标准中还没有完成用户定义相关的语法。

参见

ALTER ROLE

ALTER USER MAPPING

Name

ALTER USER MAPPING -- 更改用户映射的定义

Synopsis

```
ALTER USER MAPPING FOR { _user_name_ | USER | CURRENT_USER | PUBLIC }
    SERVER _server_name_
    OPTIONS ( [ ADD | SET | DROP ] _option_ ['_value_'] [, ... ] )
```

描述

`ALTER USER MAPPING` 更改用户映射的定义。

外服务器的所有者可以为任何用户更改那个服务器的户映射。 而且，若已将服务器的 `USAGE` 权限授予该用户，那么该用户可以为其自身的用户名修改用户映射。

参数

`_user_name_`

映射的名称。 `CURRENT_USER` 和 `USER` 匹配当前用户的名称。 `PUBLIC` 用于匹配系统中已有和将来创建的所有用户名。

`_server_name_`

用户映射的服务器名称。

`OPTIONS ( [ ADD | SET | DROP ]` `_option_` [' `_value_` '] [, ...])

用户映射更改选项。新的选项覆盖所有先前声明的选项。 `ADD` 、 `SET` 和 `DROP` 声明要执行的操作。 若未明确声明操作，则假定为 `ADD` 。 选项名称必须是独一无二的；选项也要通过服务器的外部数据封装器的验证。

例子

为用户映射 `bob` ，服务器 `foo` 更改密码:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'public');
```

兼容性

`ALTER USER MAPPING` 命令符合 ISO/IEC 9075-9(SQL/MED)标准。但是有一点点语法上的问题：标准省略了关键字 `FOR` 。 `CREATE USER MAPPING` 和 `DROP USER MAPPING` 命令都在类似的位置使用关键字 `FOR` ，并且 IBM DB2（作为其他主要SQL/MED标准的实现者） 也在 `ALTER USER MAPPING` 命令中使用这个关键字，PostgreSQL是在标准一致性和交互操作性中取了一个中间位置。

参见

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

Name

ALTER VIEW -- 更改视图定义

Synopsis

```
ALTER VIEW [ IF EXISTS ] _name_ ALTER [ COLUMN ] _column_name_ SET DEFAULT _expression_
ALTER VIEW [ IF EXISTS ] _name_ ALTER [ COLUMN ] _column_name_ DROP DEFAULT
ALTER VIEW [ IF EXISTS ] _name_ OWNER TO _new_owner_
ALTER VIEW [ IF EXISTS ] _name_ RENAME TO _new_name_
ALTER VIEW [ IF EXISTS ] _name_ SET SCHEMA _new_schema_
ALTER VIEW [ IF EXISTS ] _name_ SET ( _view_option_name_ [= _view_option_value_] [, ... ]
ALTER VIEW [ IF EXISTS ] _name_ RESET ( _view_option_name_ [, ... ] )
```

描述

`ALTER VIEW` 更改视图的各种辅助属性。 （如果你是更改视图的查询定义，要使用 `CREATE OR REPLACE VIEW` 。）

你必须是视图的所有者才可以使用 `ALTER VIEW` 。 要改变视图的模式，您必须要有新模式的 `CREATE` 权限。 要改变视图的所有者，您必须是新所属角色的直接或者 间接的成员，并且此角色必须有视图模式的 `CREATE` 权限。 （这些限制强制 更改所有者不会做任何您通过删除或者重建视图时不能做的操作。但是，一个超级用户不管怎样都可以更改任何视图的所属关系。）

参数

`_name_`

一个已有视图的名称(可以有模式修饰)。

`IF EXISTS`

使用这个选项，如果视图不存在时不会产生错误，仅有会有一个提示信息。

`SET` / `DROP DEFAULT`

这种形式设置或删除一个列的缺省值。 当 `INSERT` 和 `UPDATE` 命令的对象是视图时，使用这个选项时可以在视图相关的规则和触发器启动前，设置视图列的缺省值。 视图列的缺省值也会优先于视图相关联表的列缺省值生效。

`_new_owner_`

视图新所有者的用户名称。

`_new_name_`

视图的新名称。

`_new_schema_`

视图的新模式。

`_view_option_name_`

将要设置或复位的选项名称。

`_view_option_value_`

视图选项的新值。

注意

由于历史原因， `ALTER TABLE` 也可用于视图；但是 `ALTER TABLE` 命令中允许与视图相关的选项与上面所列选项相同。

例子

重命名视图 `foo` 为 `bar`：

```
ALTER VIEW foo RENAME TO bar;
```

对一个可更新视图增加列缺省值：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts 现在是空值
INSERT INTO a_view(id) VALUES(2);  -- ts 现在是当前时间
```

兼容性

`ALTER VIEW` 是PostgreSQL对SQL标准的扩展。

参见

CREATE VIEW, DROP VIEW

ANALYZE

Name

ANALYZE -- 收集与数据库有关的统计信息

Synopsis

```
ANALYZE [ VERBOSE ] [ _table_name_ [ ( _column_name_ [, ...] ) ] ]
```

描述

`ANALYZE` 收集数据库中表内容的统计信息，然后把结果保存在系统表 `pg_statistic` 里。 随后，查询规划器就可以使用这些统计帮助判断查询的最佳规划。

使用本命令时如果不带任何参数，`ANALYZE` 将检查当前数据库里的所有表。 如果有参数，`ANALYZE` 只检查那个指定的表。 你还可以指定一些字段的名字，在这种情况下，将只收集那些字段的统计信息。

参数

`VERBOSE`

显示处理过程的信息。

`_table_name_`

要分析的指定表(可以用模式名修饰)的名字。缺省是当前数据库里所有表（不包含外部数据表）。

`_column_name_`

要分析的指定字段的名字。缺省是所有字段。

Outputs

如果使用了 `VERBOSE` 选项，那么 `ANALYZE` 在执行过程中会显示很多进度信息，表明当前正在处理的是哪个表。 同时打印有关该表的很多其它信息。

注意

只有在明确指定了外部数据表时，这些表才会被分析处理。也不是所有的外部数据封装器都支持 `ANALYZE` 。 如果表的封装器不支持 `ANALYZE` ，在执行此命令时会显示一个警告信息，系统不会做任何处理。

在默认的PostgreSQL配置中，autovacuum守护进程（参见Section 23.1.6）负责在初次加载数据时自动分析表。 因为它们会改变整个常规操作。 当autovacuum关闭时，周期性地运行 `ANALYZE` ，或者在对表的大部分内容做了更改之后马上运行它是个好习惯。 准确的统计信息将帮助规划器选择最合适的查询规划，并因此改善查询处理的速度。 对以读取为主要负载的数据库，一种比较经常采用的策略是每天在低负荷的时候运行一次VACUUM和 `ANALYZE` 。

`ANALYZE` 只需要在目标表上有一个读取锁，因此它可以和表上的其它活动并发地运行。

`ANALYZE` 收集的统计信息通常包括每个字段最常用数值的列表以及显示每个字段里数据近似分布的柱状图。 如果 `ANALYZE` 认为它们都没有什么用(比如在一个拥有唯一约束的字段上没有公共的数值)或者是该字段数据类型不支持相关的操作符，那么它们都可以忽略。 在Chapter 23中有关于统计的更多信息。

对于大表， `ANALYZE` 采集表内容的一个随机抽样做统计， 而不是检查每一行。 这样就保证了即使是在很大的表上也只需要很少时间就可以完成分析。 不过，要注意的是统计只是近似的结果， 而且每次运行 `ANALYZE` 时，即使表的内容没有任何变化，分析的结果也可能有稍许差异。 这也会导致EXPLAIN显示的规划器的预期开销有一些小变化。 在极少的情况下，此非决定论会引发规划器在 `ANALYZE` 运行后引发查询 计划更改。 为了避免这个问题，可以提高 `ANALYZE` 收集的统计数量， 像下面描述的那样。

分析的广度可以通过用调整default_statistics_target配置参数， 或者是以每字段为基础通过用 `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS` (参见ALTER TABLE)设置每字段的统计目标来控制。 目标数值设置最常用数值列表中的记录的最大数目以及柱状图中的最大块数。 缺省的目标数值是100，不过可以调节这个数值获取规划器计算精度和 `ANALYZE` 运行所需要的时间以及 `pg_statistic` 里面占据的空间数目之间的平衡。 特别是，把统计目标设置为零就关闭了该字段的统计收集。 这么做对那些从来不参与到查询的 `WHERE` ， `GROUP BY` 或者 `ORDER BY` 选项里的字段是很有用的，因为规划器不会使用到这样的字段上的统计。

在被分析的字段中最大的统计目标决定统计采样的行数。 增大目标会导致 `ANALYZE` 的时候成比例地增大对时间和空间的需求。

`ANALYZE` 的一个估计值是出现在每列的不同值的数目。 因为仅仅行的一个子集被检查,这个估计值有时会很不准确，甚至是对最大可能的统计目标。 如果这个错误导致了差的查询计划，一个更精确的值可以通过手动确定并且然后通过 `ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)` 安装。 (参见ALTER TABLE)。

如果已分析的表有一个或者更多子表， `ANALYZE` 将会收集统计两次： 一次仅仅在父表的行上，第二次是在父表及其所有子表的行上。 第二次收集的统计数据在查询规划器遍历整个继承树结构时会用到。 不过，autovacuum守护进程在决定触发一个对一个表的自动分析时，会仅仅考虑在父表上进行插入或者更新。 如果那个表几乎不插入或者更新，继承的统计数据将不再更新，除非您手动运行 `ANALYZE` 。

如果拟分析的表成了一个空表，则 `ANALYZE` 不会记录该表的统计信息。而原来已有有统计信息则会保留。

兼容性

SQL标准里没有 `ANALYZE` 语句。

参见

VACUUM, vacuumdb, Section 18.4.4, Section 23.1.6

BEGIN

Name

BEGIN -- 开始一个事务块

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ _transaction_mode_ [, ...] ]

`_transaction_mode_`可以是以下选项之一：

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

`BEGIN` 初始化一个事务块，也就是说所有 `BEGIN` 命令后的用户语句都将在一个事务里面执行直到给出一个明确的COMMIT或ROLLBACK。 缺省情况下(没有 `BEGIN`)，PostgreSQL以"autocommit"模式执行事务，也就是说，每个语句在其自身的事务中执行，并且在语句结束的时候执行了一次隐含的提交。 （如果执行成功则隐含地执行一个提交，否则执行回滚。）

在事务块里语句执行的明显快得多，因为事务开始/提交需要大量的CPU和磁盘活动。 在一个事务内部执行多条语句对于修改若干个相关的表的时候也是很有用的： 在所有相关的更新完成之前，其它会话看不到中间的状态。

如果指定了隔离级别、读/写模式或者是延迟模式，那么新事务将具有那些特征。 就像执行了SET TRANSACTION一样。

参数

`WORK``TRANSACTION`

可选关键字，没什么作用。

参考SET TRANSACTION获取这个语句的其它参数的含义。

注意

START TRANSACTION有着和 `BEGIN` 一样的功能。

使用COMMIT或ROLLBACK结束一个事务块。

在一个现有事务块内部发出一个 `BEGIN` 将产生一个警告信息。 事务的状态将不会被影响。 要想在一个事务块里嵌套事务，请使用保存点功能(参见SAVEPOINT)。

出于向下兼容考虑，在随后的 `_transaction_modes_` 之间的逗号可以忽略 。

例子

开始一个事务块：

```
BEGIN;
```

兼容性

`BEGIN` 是PostgreSQL语言的扩展。 它等价于SQL标准中的START TRANSACTION命令，在其命令的资料中包含更多有关兼容性的信息。

`DEFERRABLE``_transaction_mode_` 选项是PostgreSQL的扩展。

顺便说一句， `BEGIN` 关键字在嵌入SQL里用于不同目的。建议你在移植数据库应用时仔细检查事务的语意。

参见

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CHECKPOINT

Name

CHECKPOINT -- 强制一个事务日志检查点

Synopsis

```
CHECKPOINT
```

描述

检查点是一个事务日志中的时点，所有数据文件都在该时点被更新以反映日志中的信息，所有数据文件都将被刷新到磁盘。 请参见Section 29.4获取更多有关的信息。

`CHECKPOINT` 命令执行时会强制设置一个检查点，而不必等待正常的系统内部按调度程序设置的检查点（其设置见Section 18.5.2）。 `CHECKPOINT` 在正常的操作中一般不必使用。

在数据恢复过程中， `CHECKPOINT` 命令会强制设置一个复位点（参见Section 29.4），而不是设置一个检查点。

只有超级用户可以调用 `CHECKPOINT` 。

兼容性

`CHECKPOINT` 命令是PostgreSQL语言的扩展。

CLOSE

Name

CLOSE -- 关闭游标

Synopsis

```
CLOSE { _name_ | ALL }
```

描述

`CLOSE` 释放和一个游标关联的所有资源。不允许对一个已关闭的游标再做任何操作。一个不再使用的游标应该尽早关闭。

当创建游标的事务用 `COMMIT` 或 `ROLLBACK` 终止之后，每个不可保持的已打开游标都隐含关闭。 当创建游标的事务通过 `ROLLBACK` 退出之后，每个可以保持的游标都将隐含关闭。 当创建游标的事务成功提交，那么可保持的游标保持打开，直到执行一个明确的 `CLOSE` 命令或者客户端断开。

参数

`_name_`

一个待关闭的游标的名字。

`ALL`

关闭所有打开的游标。

注意

PostgreSQL没有明确打开游标的 `OPEN` 语句； 一个游标在使用DECLARE 语句声明的时候就可以被认为是打开了。 可以使用DECLARE命令声明游标。

你可以通过查询 `pg_cursors` 系统视图查看所有可获得的游标。

如果一个游标在一个随后回滚的保存点后关闭， `CLOSE` 不会回滚； 这就是说，游标仍然关闭。

例子

关闭游标 `liahona`：

```
CLOSE liahona;
```

兼容性

`CLOSE` 命令与SQL标准完全兼容。 `CLOSE ALL` 是一个PostgreSQL扩展。

参见

DECLARE, FETCH, MOVE

CLUSTER

Name

CLUSTER -- 根据一个索引对某个表盘簇化排序

Synopsis

```
CLUSTER [VERBOSE] _table_name_ [ USING _index_name_ ]
CLUSTER [VERBOSE]
```

描述

`CLUSTER` 指示PostgreSQL基于索引 `_index_name_` 的内容对表 `_table_name_` 进行存储盘簇化排序。 索引必须已经在表 `_table_name_` 上定义过的索引。

当对一个表盘簇化排序后，该表的物理存储将基于索引顺序排序。盘簇化是一次性操作：当表将来被更新之后，更改的内容不会被盘簇化排序。 也就是说，系统不会试图按照索引顺序对更新过的记录重新盘簇化排序。 （如果用户想要这个效果，可以通过周期性地手工执行该命令的方法重新盘簇化排序。 并且，设置表的 `FILLFACTOR` 存储参数为小于100%可以帮助盘簇化排序在更新时排序，因为若有足够的空间可用，更新的行可以保存在相同的页面位置。）

在对一个表盘簇化排序之后，PostgreSQL会记忆使用了哪个索引上进行了盘簇化排序。 `CLUSTER` `_table_name_` 的形式在表以前进行盘簇化排序的同一个索引上重新盘簇化排序。 也可以用 `CLUSTER` 或 `SET WITHOUT CLUSTER` 形式来设置用于进行盘簇化排序的索引，或清除任何之前的设置。

不含参数的 `CLUSTER` 会将当前用户所拥有的当前数据库中的所有先前进行盘簇化排序的表重新处理，或者如果是超级用户使用这个命令时，则对所有进行过盘簇化排序表重新处理。 这种形式的 `CLUSTER` 不能在一个事务里面调用。

在对一个表进行盘簇化排序的时候，会在其上请求一个 `ACCESS EXCLUSIVE` 锁。这样就避免了在 `CLUSTER` 完成之前执行任何其它的数据库操作(包括读写)。

参数

`_table_name_`

表的名称（可以有模式修饰）。

`_index_name_`

一个索引名称。

`VERBOSE`

当每一个表进行盘簇化排序时打印一个处理情况报告。

注意

如果你只是随机的访问表中的行，那么表中数据的实际存储顺序是无关紧要的。 但是，如果对某些特定数据的访问较多，而且有一个索引将这些数据分组，那么使用 `CLUSTER` 会非常有益处。 如果从一个表中请求一定索引范围的值，或者是一个索引值对应多行， `CLUSTER` 也会有助于应用，因为如果索引标识出第一匹配行所在的存储页，所有其它行也可能已经在同一个存储页里了，这样便节省了磁盘访问的时间，加速了查询。

`CLUSTER` 在盘簇化排序的处理过程中，可以按一个索引顺序，也可以按一个排序后的顺序扫描内容。 它会基于查询规划器的成本参数和表的统计信息选择一个相对较快的方法。

`CLUSTER` 在盘簇化排序的处理过程中，系统先创建一个按照索引顺序建立的表的临时拷贝。 同时也建立表上的每个索引的临时拷贝。 因此，需要磁盘上有足够的剩余空间，至少是表大小和索引大小的和。

当使用顺序扫描和排序操作时，系统会创建临时排序文件，这样极端情况下，磁盘空间会需要至少约2倍的表大小和索引大小。 这个方法一般比使用索引的方法要快一点，但如果对磁盘空间的要求不可接受，可以临时设置enable_sort 为 `off` 来禁用这个选择。

建议在执行盘簇化排序前，将maintenance_work_mem参数设置为一个合理的较大数值（但不要超过可以保留给 `CLUSTER` 使用的内存大小）。

因为规划器记录着有关表的排序的统计，所以建议在最近盘簇化排序后的表上运行ANALYZE。否则，规划器可能会选择很差劲的查询规划。

因为 `CLUSTER` 记录着哪些索引用于过盘簇化排序，所以用户可以第一次手工指定表使用指定索引进行盘簇化排序，以后设置一个周期化执行的维护脚本，只需执行不带参数的 `CLUSTER` 命令，即可实现对想要周期性盘簇化排序的表进行自动更新。

例子

按照索引 `employees_ind` 的顺序对 `employees` 表进行盘簇化排序：

```
CLUSTER employees USING employees_ind;
```

使用以前用过的同一个索引对 `employees` 表进行盘簇化排序：

```
CLUSTER employees;
```

对以前盘簇化排序过的所有表进行重新盘簇化排序：

```
CLUSTER;
```

兼容性

SQL标准里没有 `CLUSTER` 语句。

```
 CLUSTER _index_name_ ON _table_name_
```

的语法也兼容PostgreSQL 8.3之前的版本

参见

clusterdb

COMMENT

Name

COMMENT -- 定义或者改变一个对象的注释

Synopsis

```
COMMENT ON
{
  AGGREGATE _agg_name_ (_agg_type_ [, ...] ) |
  CAST (_source_type_ AS _target_type_) |
  COLLATION _object_name_ |
  COLUMN _relation_name_._column_name_ |
  CONSTRAINT _constraint_name_ ON _table_name_ |
  CONVERSION _object_name_ |
  DATABASE _object_name_ |
  DOMAIN _object_name_ |
  EXTENSION _object_name_ |
  EVENT TRIGGER _object_name_ |
  FOREIGN DATA WRAPPER _object_name_ |
  FOREIGN TABLE _object_name_ |
  FUNCTION _function_name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] ) |
  INDEX _object_name_ |
  LARGE OBJECT _large_object_oid_ |
  MATERIALIZED VIEW _object_name_ |
  OPERATOR _operator_name_ (_left_type_, _right_type_) |
  OPERATOR CLASS _object_name_ USING _index_method_ |
  OPERATOR FAMILY _object_name_ USING _index_method_ |
  [ PROCEDURAL ] LANGUAGE _object_name_ |
  ROLE _object_name_ |
  RULE _rule_name_ ON _table_name_ |
  SCHEMA _object_name_ |
  SEQUENCE _object_name_ |
  SERVER _object_name_ |
  TABLE _object_name_ |
  TABLESPACE _object_name_ |
  TEXT SEARCH CONFIGURATION _object_name_ |
  TEXT SEARCH DICTIONARY _object_name_ |
  TEXT SEARCH PARSER _object_name_ |
  TEXT SEARCH TEMPLATE _object_name_ |
  TRIGGER _trigger_name_ ON _table_name_ |
  TYPE _object_name_ |
  VIEW _object_name_
} IS '_text_'
```

描述

`COMMENT` 存储一个数据库对象的注释。

每个对象只存储一条注释，因此要修改一个注释，对同一个对象发出一条新的 `COMMENT` 命令即可。 要删除注释，在文本字符串的位置写上 `NULL` 即可。 当删除对象时，注释自动被删除掉。

对大多数对象，只有对象的所有者可以设置注释。角色没有所有者，所以 `COMMENT ON ROLE` 命令仅可以由超级用户对超级用户角色执行， 有 `CREATEROLE` 权限的角色也可以为非超级用户角色设置注释，当然超级用户可以对所有对象进行注释。

注释可以用psql程序中的 `\d` 命令检索。 其它希望提取注释的用户接口设计可以使用程序psql使用的同样的内置函数 `obj_description` 、 `col_description` 和 `shobj_description` (参见Table 9-55)。

参数

`_object_name_``_relation_name_` . `_column_name_` `_agg_name_` `_constraint_name_` `_function_name_` `_operator_name_` `_rule_name_` `_trigger_name_`

要加入注释的对象名称。表、聚集、排序规则、编码转换、域、外部表、函数、索引、操作符、操作符类、操作符系列、序列、全文搜索对象 、类型、视图，名字可以有模式修饰。 当对一个字段进行注释时， `_relation_name_` 必需是针对一个表、视图、复合类型或是外部表。

`_agg_type_`

聚集函数操作的输入参数类型，要引用一个零参数聚集函数，可以使用 `*` 代替输入数据类型列表。

`_source_type_`

类型转换的源数据类型 。

`_target_type_`

类型转换的目标数据类型 。

`_argmode_`

函数参数的模式： `IN` 、 `OUT` 、 `INOUT` 或 `VARIADIC` 。如果省略，缺省值是 `IN` 。 请注意 `COMMENT ON FUNCTION` 实际上不会使用 `OUT` 参数， 因为只要有输入参数就可以判断函数的身份了。 因此，只要列出 `IN` 、 `INOUT` 和 `VARIADIC` 参数就足够了。

`_argname_`

函数参数的名字。请注意 `COMMENT ON FUNCTION` 实际上并不使用参数名，因为只要有参数的数据类型就可以判断函数的身份。

`_argtype_`

如果有的话，是函数参数的数据类型(可以用模式修饰)

`_large_object_oid_`

大对象的OID 。

`_left_type_` `_right_type_`

操作符参数的数据类型（可以用模式修饰）。 当前置或后置操作符不存在时，可以增加 `NONE` 选项。

`PROCEDURAL`

这个选项没有任何用处。

`_text_`

新的注释，以字符串文本的方式写；如果是 `NULL` 则删除注释。

注意

目前注释浏览没有安全机制：任何连接到某数据库上的用户都可以看到所有该数据库对象的注释。 共享对象(比如数据库、角色、表空间)的注释是全局存储的，连接到任何数据库的任何用户都可以看到它们。 因此，不要在注释里存放与安全有关的敏感信息。

例子

给表 `mytable` 加注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

再删除注释：

```
COMMENT ON TABLE mytable IS NULL;
```

其它一些例子：

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for swedish language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

兼容性

SQL标准里没有 `COMMENT` 命令。

COMMIT

Name

COMMIT -- 提交当前事务

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

描述

`COMMIT` 提交当前事务。 所有事务的更改都将被其它事务可见， 而且保证即使有崩溃发生时的数据有效性。

参数

`WORK``TRANSACTION`

可选关键字，没有任何作用。

注意

使用ROLLBACK语句退出事务。

在一个事务外部发出 `COMMIT` 不会有问题，但是将产生一个警告信息。

例子

提交当前事务以让所有变更永久化：

```
COMMIT;
```

兼容性

SQL标准只声明了 `COMMIT` 和 `COMMIT WORK` 两种形式。 否则这条命令与标准完全兼容。

参见

BEGIN, ROLLBACK

COMMIT PREPARED

Name

COMMIT PREPARED -- 提交一个早先为两阶段提交准备好的事务

Synopsis

```
COMMIT PREPARED _transaction_id_
```

描述

`COMMIT PREPARED` 提交一个早先为两阶段提交准备好的事务。

参数

`_transaction_id_`

要提交的事务标示符。

注意

要提交一个预处理的事务，你必须是最初执行该事务的用户或超级用户。 不过你不必在同一个会话里执行该命令。

这条命令不能在事务块里执行。预处理的事务立即提交。

所有目前可用的预处理事务都在系统视图 `pg_prepared_xacts` 里列出。

Examples

提交事务标识符为 `foobar` 的事务:

```
COMMIT PREPARED 'foobar';
```

兼容性

`COMMIT PREPARED` 是PostgreSQL的扩展。 该命令设计给外部事务管理系统使用的，SQL标准只涉及部分功能（如X/Open XA），但这些功能目前没有标准化。

参见

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

Name

COPY -- 在表和文件之间拷贝数据

Synopsis

```
COPY _table_name_ [ ( _column_name_ [, ...] ) ]
    FROM { '_filename_' | PROGRAM '_command_' | STDIN }
    [ [ WITH ] ( _option_ [, ...] ) ]

COPY { _table_name_ [ ( _column_name_ [, ...] ) ] | ( _query_ ) }
    TO { '_filename_' | PROGRAM '_command_' | STDOUT }
    [ [ WITH ] ( _option_ [, ...] ) ]

where `_option_` can be one of:

    FORMAT _format_name_
    OIDS [ _boolean_ ]
    FREEZE [ _boolean_ ]
    DELIMITER '_delimiter_character_'
    NULL '_null_string_'
    HEADER [ _boolean_ ]
    QUOTE '_quote_character_'
    ESCAPE '_escape_character_'
    FORCE_QUOTE { ( _column_name_ [, ...] ) | * }
    FORCE_NOT_NULL ( _column_name_ [, ...] )
    ENCODING '_encoding_name_'
```

描述

`COPY` 在PostgreSQL表和文件之间交换数据。 `COPY TO` 把一个表的所有内容都拷贝到一个文件，而 `COPY FROM` 从一个文件里拷贝数据到一个表里(把数据附加到表中已经存在的内容里)。 `COPY TO` 还能拷贝 `SELECT` 查询的结果。

如果声明了一个字段列表， `COPY` 将只在文件和表之间拷贝已声明字段的数据。 如果表中有任何不在字段列表里的字段，那么 `COPY FROM` 将为那些字段插入缺省值。

带文件名的 `COPY` 指示PostgreSQL服务器直接从文件中读写数据。 如果声明了文件名，那么服务器必须可以访问该文件，而且文件名必须从服务器的角度声明。 如果使用了 `PROGRAM` 选项，则服务器会从指定的这个程序进行输入或是写入该程序作为输出。 如果使用了 `STDIN` 或 `STDOUT` 选项，那么数据将通过客户端和服务器之间的连接来传输。

参数

`_table_name_`

现存表的名字(可以有模式修饰)。。

`_column_name_`

可选的待拷贝字段列表。如果没有声明字段列表，那么将使用所有字段。

`_query_`

一个必须用圆括弧包围的SELECT或VALUES命令，其结果将被拷贝。

`_filename_`

输入或输出文件的路径名。输入文件名可以是绝对或是相对的路径，但输出文件名必须是绝对路径。 Windows用户可能需要使用 `E''` 字符串和双反斜线作为路径名称。

`PROGRAM`

需执行的程序名。在 `COPY FROM` 命令中，输入是从程序的标准输出中读取，而在 `COPY TO` 中，命令的输出会作为程序的标准输入。

注意，程序一般是在命令行界面下执行，当用户需要传递一些变量给程序时，如果这些变量的来源不是可靠的，用户必须小心过滤处理那些对命令行界面来说是有特殊意义的字符。 基于安全的原因，最好是使用固定的命令字符串，或者至少是应避免直接使用用户输入（应先过滤特殊字符）。

`STDIN`

声明输入是来自客户端应用。

`STDOUT`

声明输入将写入客户端应用。

`_boolean_`

声明用户所选的选项是否应该被开启或者关闭。您可以写 `TRUE` 、 `ON` 或 `1` 来启用这个选项，并且用 `FALSE` 、 `OFF` 或 `0` 来关闭它。 `_boolean_` 值也可以被省略，此时系统使用缺省值 `TRUE` 。

`FORMAT`

选择被读或者写的数据格式： `text` 、 `csv` （逗号分隔值），或者 `binary` 。 默认是 `text` 。

`OIDS`

声明为每行记录都拷贝内部对象标识(OID)。 （如果为一个 `_query_` 拷贝或者没有 `OIDS` 的表声明了OIDS选项，则抛出一个错误。）

`FREEZE`

请求拷贝那些已冻结的数据，就类似使用 `VACUUM FREEZE` 的效果。 这主要于用于初始化时加载数据时的性能考虑。 仅在表记录初始创建或是在当前子事务中被清理的记录会补冻结，没有游标会打开，事务中也没有数据快照。

注意此时其他的事务会立刻看见刚加载的数据。 这不符合MVCC正常的可见性规则，用户应注意这可能带来的潜在问题。

`DELIMITER`

指定分隔每一行记录中的列的字符。默认是文本格式的制表符， `CSV` 格式的逗号。 必须有一个独立的一字节的字符。 在使用 `binary` 格式时这个选项是不允许的。

`NULL`

声明代表一个空值的字符串。默认是文本格式的 `\N` ， `CSV` 格式的一个未被引用的空字符串。 即使是文本格式您可能也更偏向于空串，例如您不想从空字符串中区分空值。 在使用 `binary` 格式时这个选项是不允许的。

> **Note:** 在使用 `COPY FROM` 的时候，任何匹配这个字符串的字符串将被存储为NULL值，所以你应该确保你用的字符串和 `COPY TO` 相同。

`HEADER`

声明文件包含一个带有文件中每列名称的标题行。 在输出时，第一行包含表中的列名，在输入时，第一行是被忽略的。该选项仅仅在使用 `CSV` 格式时是允许的。

`QUOTE`

指定引用数据的引用字符。默认的是双引号。这一定是一个1字节的字符。该选项仅仅在使用 `CSV` 格式时允许。

`ESCAPE`

声明应该出现在一个匹配 `QUOTE` 值的数据字符之前的字符。 默认与 `QUOTE` 值相同（所以若它出现在数据中，则引用字符是翻一倍）。 这一定是一个1字节的字符。该选项只有在使用 `CSV` 格式时允许。

`FORCE_QUOTE`

强制引用在每个指定列的所有非 `NULL` 值。 `NULL` 从不被引用。 如果声明了 `*` ，非 `NULL` 值将在所有列中被引用。 这个选项仅仅在 `COPY TO` 中并且仅仅在使用 `CSV` 格式时允许。

`FORCE_NOT_NULL`

默认情况下空字符串是空的，这意味着空值将会被读作长度为零的字符串而不是空值，即使当他们不被引用。 这个选项仅仅在 `COPY FROM` 中并且仅仅在使用 `CSV` 格式时允许。

`ENCODING`

声明文件的编码集是 `_encoding_name_` 。 如果这个选项省略，则系统使用当前的用户编码集。阅读下面的注意事项以了解更多内容。

Outputs

当 `COPY` 命令执行成功后，会在屏幕上显示

```
COPY _count_
```

式样内容， 这里 `_count_` 是已拷贝成功的记录数。

注意

`COPY` 只能用于表，不能用于视图。当然也可以用于 `COPY (SELECT * FROM` `_viewname_`) TO ...

`COPY` 仅仅处理已指定的特定表；它将不复制数据到子表或从子表中复制数据。 因此比如 `COPY` `_table_` TO显示与 `SELECT * FROM ONLY` `_table_` 相同的数据。 但是 `COPY (SELECT * FROM` `_table_`) TO ...可以用于转储在继承层次结构的所有数据。

你对任何要 `COPY TO` 出来的数据必须有查询的权限， 对任何要 `COPY FROM` 入数据的表必须有插入权限。 对列在命令中的字段拥有列权限也是必须的。

`COPY` 命令里面的文件必须是由服务器直接读或写的文件，而不是由客户端应用读写。 因此，它们必须位于数据库服务器上或者可以被数据库服务器所访问，而不是客户端程序。 它们必须被运行PostgreSQL服务器的用户可读或写，而不是客户端程序。 由 `PROGRAM` 选项指定的命令必须是由服务器来执行的，而不是客户端程序，必须是由PostgreSQL所属的用户。 `COPY` 在指定一个程序或是命令时只允许数据库超级用户来执行，因为它允许读写任意服务器有权限访问的文件。

不要混淆 `COPY` 和 psql应用程序中的 `[\copy](#calibre_link-1638)` 指令。 `\copy` 调用 `COPY FROM STDIN` 或 `COPY TO STDOUT` ， 然后把数据抓取/存储到一个psql客户端可以访问的文件中。 因此，使用 `\copy` 的时候，文件访问权限是由客户端应用程序而不是服务器端决定的。

建议在 `COPY` 里的文件名字总是使用绝对路径。 在 `COPY TO` 的时候是由服务器强制进行的，但是对于 `COPY FROM` ，你的确可从一个相对路径的文件里读取。 该路径将解释为相对于服务器的工作目录(通常是数据目录)，而不是客户端的工作目录。

执行一个 `PROGRAM` 选项指定的命令有可能还会受到操作系统的存取权限控制，如在SELinux下。

`COPY FROM` 在执行时会触发目标表上所有触发器和检查约束。不过，不会执行规则。

`COPY` 输入和输出会被 `DateStyle` 参数影响。 为了和其它PostgreSQL不同服务器间进行数据转移(它们可能是非缺省 `DateStyle` 设置)，应该在使用 `COPY TO` 前把 `DateStyle` 参数值设置为 `ISO` 。 另外也建议在导出数据时，不要将 `IntervalStyle` 参数设置为 `sql_standard` 。 因为负的区间值可能会被对 `IntervalStyle` 有不同设置的服务器误解。

输入数据通过`ENCODING`参数或是当前客户端编码来解译，输出数据也是通过`ENCODING`参数或是为当前客户端的编码来编码， 即使数据不经过客户端的，仍会通过服务器直接将数据从文件中读出或者写入到文件中去。

`COPY`在第一个错误处停下来。 这些在`COPY TO`中不应该导致问题，但在`COPY FROM`时目标表会已经接收到早先的行， 这些行将不可见或不可访问，但是仍然会占据磁盘空间。如果你碰巧拷贝大量数据文件的话，这些东西积累起来可能会占据相当大的磁盘空间。 你可以调用`VACUUM`来恢复那些磁盘空间。

文件格式

文本格式

当使用`text`格式时，读写的文件是一个文本文件，每行代表表中一个行。 行中的列(字段)用分隔符分开。字段值本身是由与每个字段类型相关的输出函数生成的字符串，或者是输入函数可接受的字符串。 数据中使用特定的NULL字符串表示那些值为NULL的字段。 如果输入文件的任意行包含比预期多或者少的字段，那么`COPY FROM`将抛出一个错误。 如果声明了`OIDS`选项，那么OID将作为第一个字段读写，放在所有用户字段前面。

数据的结束可以用一个只包含反斜杠和句点(`\.`)的行表示。 如果从文件中读取数据，那么数据结束的标记是不必要的，因为文件结束符可以起到相同的作用； 但是在3.0之前的客户端协议里， 如果在客户端应用之间拷贝数据，那么必须要有结束标记。

反斜杠字符(`\`)可以用于`COPY`数据，来引用那些可能会被当作行或列分隔符的数据字符。 特别是以下字符，若以一列值的一部分出现则必须在前面加上反斜杠：反斜杠、换行符、回车以及当前的分隔符字符。

声明的空字符串被`COPY TO`不加任何反斜杠发送； 与之相对，`COPY FROM`在删除反斜杠之前拿它的输入与空字符串比较。 因此，像`\N`这样的空字符串不会和实际数据值`\N`之间混淆(因为后者会表现成`\\N`)。

`COPY FROM`能够识别下列特殊反斜杠字符：

字符形式	字符含义
`\b`	反斜杠 (ASCII 8)
`\f`	进纸 (ASCII 12)
`\n`	换行符 (ASCII 10)
`\r`	回车符 (ASCII 13)
`\t`	水平制表符 (ASCII 9)
`\v`	垂直制表符 (ASCII 11)
`` \``_digits_ ``	反斜杠后面跟着一到三个八进制数，表示ASCII值为该数的字符
`` \x``_digits_ ``	反斜杠 `\x` 后面跟着一个或两个十六进制位声明指定数值编码的字符

目前， `COPY TO` 绝不会发出一个八进制或者十六进制反斜杠序列，但是它的确使用了上面列出的其它字符用于控制字符。

任何其他未在上表中提及的斜字符将会用来表示其本身。然而，也要注意不必要的情况添加反斜杠。 因为这可能意外地生成一个匹配数据结束标记(`\.`)或者空字符串 (默认为 `\N` ）的字符串。 这些字符串将在任何其他反斜杠处理做完之前确认。

强烈建议产生 `COPY` 数据的应用程序将数据换行符和回车分别转换为 `\n` 和 `\r` 序列。 目前，可以由反斜杠和回车代表一个数据回车，并且由反斜杠和换行符代表一个数据换行。 然而，这些表示法在将来的版本中可能无法接受。 `COPY` 文件在不同操作系统之间转移时，它们也非常容易被误解读， （例如：从Unix 系统移到Windows系统，或者反过来）。

`COPY TO` 将在每行的结尾用一个Unix风格的换行符(" `\n` ")。 运行在Windows上的服务器会输出的回车换行符(" `\r\n` ")，但只是用于 `COPY` 到服务器 文件里； 为了在不同平台之间一致， `COPY TO STDOUT` 总是发送" `\n` "而不管服务器平台是什么。 `COPY FROM` 可以处理那些以回车符、换行符、回车/换行符作为行结束的数据。 为了减少在数据中出现的未逃逸的新行或者回车导致的错误，如果输入的行结尾不像上面这些符号， `COPY FROM` 会发出警告。

CSV 格式

这个格式用于输入和输出逗号分隔数值(`CSV`)文件格式，许多其它程序都用这个文件格式，比如电子表格。 这个模式下生成并识别逗号分隔的CSV逃逸机制，而不是使用PostgreSQL标准文本的逃逸模式。

每条记录的值都是用 `DELIMITER` 字符分隔的。 如果数值本身包含分隔字符、 `QUOTE` 字符、 `NULL` 字符串、回车符、换行符，那么整个数值用 `QUOTE` 字符前缀和后缀(包围)， 并且数值里任何 `QUOTE` 字符或 `ESCAPE` 字符都前导逃逸字符。 你也可以使用 `FORCE_QUOTE` 在输出非 `NULL` 的指定字段值时强制引号包围。

`CSV` 格式没有标准的办法区分一个 `NULL` 值和一个空字符串。 PostgreSQL的 `COPY` 通过引号包围来处理这些。 一个当作 `NULL` 输出的 `NULL` 参数值是没有引号包围的，而匹配非 `NULL` 字符串的参数值是用引号包围的。 比如，使用缺省设置时，一个 `NULL` 是写做一个无引号包围的空字符串，而一个空字符串数值写做 双引号包围(`""`)。 读取数值也遵循类似的规则。 你可以使用 `FORCE_NOT_NULL` 来避免为特定字段进行 `NULL` 比较。

因为对于 `CSV` 格式而言，反斜杠不是特殊字符，数据的结束标志(`\.`) 可以作为数据值出现。 为了避免任何可能的歧意，一个单独的 `\.` 数据值在输出中将被自动使用引号包围 ；在输入中，如果被引号界定，那么将不会当作数据结束标志。如果你要加载其它程序创建的、有未用引号界定字段的文件，并且其中含有 `\.` 值，你就必须用引号进行界定。

> **Note:** 在 `CSV` 模式下，所有字符都是有效的。 一个被空白包围的引号界定数值， 或者任何非 `DELIMITER` 字符，都会被包含这些字符。 如果你给 `CSV` 行填充空白的系统里导入数据到定长字段，那么可能会导致错误。 如果出现这种情况，你可能需要先 处理一下 `CSV` 文件， 删除结尾空白，然后再向PostgreSQL里导入数据。
>
> **Note:** CSV格式可以识别和生成引号包围的回车和换行的CSV文件。因此这些文件并不像文本模式的文件那样严格地每条记录一行。
>
> **Note:** 许多程序生成奇怪的并且有时候不正确的CSV文件，所以这个文件格式更像一种惯用格式， 而不是一种标准。 因此你可能碰到一些不能使用这个机制输入的文件， 而 `COPY` 也可能生成一些其它程序不能处理的文件。

二进制格式

`binary` 形式的选项会使得所有的数据被存储/读作二进制格式而不是文本。 这比文本和 `CSV` 格式的要快一些，但是一个二进制格式文件在机器架构和PostgreSQL版本之间的可移植性比较差。 另外，二进制格式是对数据类型有一定要求的；例如，不能从 `smallint` 列中输出二进制数据 并将二进制数据读入 `integer` 列，尽管在文本格式下那会运行良好。

`binary` 文件格式包含一个文件头，0或更多包含行数据的元组，以及一个文件尾。头和数据按照网络字节顺序。

> **Note:** 7.4版本之前的PostgreSQL版本使用的是不同的二进制文件格式。

文件头

文件头由15个字节的固定域组成，后面跟着一个变长的头扩展区。固定域是：

签名

11字节的序列 `PGCOPY\n\377\r\n\0` —，请注意字节零是签名必须的一部分。 （使用这个签名是为了能够很容易看出文件是否已经被一个非8位安全的转换器给破坏了。 这个签名会被行尾转换过滤器、删除字节零、删除高位、奇偶变化而改变。）

标志域

32位整数掩码表示该文件格式的重要方面。 位是从 0(LSB)到 31(MSB) 编码的，请注意这个域是以网络字节顺序存储的(高位在前)，后继的整数都是如此。 位16-31是 保留用做关键文件格式信息的；如果阅读器发现一个不认识的位出现在这个范围内，那么它应该退出。 位0-15都保留为标志向后兼容的格式使用；阅读器可以忽略这个范围内的不认识的位。 目前只定义了 一个标志位，而其它的必须是零：

Bit 16

如果为1，那么在数据中包括了OIDS；如果为0，则没有。

头扩展范围长度

32位整数，以字节计的头剩余长度，不包括自身。 目前，它是零，后面紧跟第一条记录行。 对该格式的更多修改都将允许额外的数据出现在头中。 阅读器应该忽略任何它不知道该如何处理的头扩展数据。

头扩展数据用来保存自定义数据序列块。 这个标志域无意告诉阅读器扩展区的内容是什么。 头扩展的具体设计内容留给以后的版本使用。

这样设计就允许向后兼容的头扩展(增加头扩展块或设置低位序标志位)以及非向后兼容的修改(设置高位标志位以标识这样的修改，并且根据需要向扩展区域增加支持数据)。

行记录

每条行都以一个16位整数计数开头，该计数是行中字段的数目(目前，在一个表里的每行都有相同的计数，但可能不会永远这样)。 然后后面不断出现行中的各个字段，字段先是一个32位的长度字，后面跟着很多的字段数据。长度字并不包括自己，并且可以为零。 一个特例是：-1表示一个NULL字段值。在NULL情况下，后面不会跟着数值字节。

在数据域之间没有对齐填充或者任何其它额外的数据。

目前，一个二进制格式文件里的所有数据值都假设是二进制格式的(格式代码为一)。 预计将来的扩展可能增加一个头域，允许为每个字段声明格式代码。

为了判断实际行数据的正确二进制格式，你应该阅读PostgreSQL源代码，特别是该字段数据类型的 `*send` 和 `*recv` 函数 (这些函数可以在源代码的 `src/backend/utils/adt/` 目录找到)。

如果在文件中包括了OIDs，那么该OID域立即跟在字段计数字后面。 它是一个普通的字段，只不过它没有包括在字段计数。 但它包括长度字，这样就允许方便的处理4字节和8字节的OIDs，并且如果某个家伙允许OIDs是可选的话，那么还可以把OIDs显示成空。

文件尾

文件尾包括保存着-1的一个16位整数字。这样就很容易与一条行的域计数字相区分。

如果一个域计数字既不是-1也不是预期的字段的数目，那么阅读器应该报错。这样就提供了对丢失与数据同步的额外检查。

例子

下面的例子把一个表拷贝到客户端，使用竖线(`|`)作为域分隔符：

```
COPY country TO STDOUT (DELIMITER '|');
```

从文件中拷贝数据到 `country` 表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

把'A'开头的国家名拷贝到一个文件里：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_
```

可以通过将输出数据通过管道方式重定向至一个外部压缩程序的方式将数据拷贝至一个压缩文件中：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

下面是一个可以从 `STDIN` 中拷贝数据到表中的例子：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意，每行里的空白实际上是一个水平制表符。

下面的是同样的数据，以二进制形式输出。 这些数据是用Unix工具 `od -c` 过滤之后输出的。 该表有三个字段；第一个是 `char(2)` ，第二个是 `text` ，第三个是 `integer` 。 所有的行在第三个域都是一个null值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

兼容性

SQL标准里没有 `COPY` 语句。

PostgreSQL9.0以前使用下面的语法，现在仍然支持：

```
COPY _table_name_ [ ( _column_name_ [, ...] ) ]
    FROM { '_filename_' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] '_delimiter_' ]
          [ NULL [ AS ] '_null string_' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] '_quote_' ]
                [ ESCAPE [ AS ] '_escape_' ]
                [ FORCE NOT NULL _column_name_ [, ...] ] ] ]

COPY { _table_name_ [ ( _column_name_ [, ...] ) ] | ( _query_ ) }
    TO { '_filename_' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] '_delimiter_' ]
          [ NULL [ AS ] '_null string_' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] '_quote_' ]
                [ ESCAPE [ AS ] '_escape_' ]
                [ FORCE QUOTE { _column_name_ [, ...] | * } ] ] ]
```

请注意：在这个语法中， `BINARY` 和 `CSV` 是作为独立的关键字，而不是作为 `FORMAT` 选项的一个参数。

PostgreSQL7.3以前使用下面的语法，现在仍然支持：

```
COPY [ BINARY ] _table_name_ [ WITH OIDS ]
    FROM { '_filename_' | STDIN }
    [ [USING] DELIMITERS '_delimiter_' ]
    [ WITH NULL AS '_null string_' ]

COPY [ BINARY ] _table_name_ [ WITH OIDS ]
    TO { '_filename_' | STDOUT }
    [ [USING] DELIMITERS '_delimiter_' ]
    [ WITH NULL AS '_null string_' ]
```

CREATE AGGREGATE

Name

CREATE AGGREGATE -- 定义一个新的聚集函数

Synopsis

```
CREATE AGGREGATE _name_ ( _input_data_type_ [ , ... ] ) (
    SFUNC = _sfunc_,
    STYPE = _state_data_type_
    [ , FINALFUNC = _ffunc_ ]
    [ , INITCOND = _initial_condition_ ]
    [ , SORTOP = _sort_operator_ ]
)

or the old syntax

CREATE AGGREGATE _name_ (
    BASETYPE = _base_type_,
    SFUNC = _sfunc_,
    STYPE = _state_data_type_
    [ , FINALFUNC = _ffunc_ ]
    [ , INITCOND = _initial_condition_ ]
    [ , SORTOP = _sort_operator_ ]
)
```

描述

`CREATE AGGREGATE` 定义一个新的聚集函数。 一些常用的聚集函数已经包含在基础软件包里了；在Section 9.20里有文档说明。 如果你需要定义一个新类型或需要一个还没有提供的聚集函数，这时 `CREATE AGGREGATE` 便可派上用场。

如果给出了一个模式的名字(比如 `CREATE AGGREGATE myschema.myagg ...`)，那么该聚集函数是在指定模式中创建的。 否则它是在当前模式中创建的。

一个聚集函数是用它的名字和输入数据类型来标识的。 同一模式中如果两个聚集处理的输入数据不同，它们可以有相同的名字。 一个聚集函数的输入数据类型必须和所有同一模式中的普通函数的名字和输入 类型不同。

一个聚集函数是用一个或两个普通函数组成的：一个状态转换函数 `_sfunc_` 和一个可选的最终计算函数 `_ffunc_` 。 它们是这样使用的：

```
_sfunc_( internal-state, next-data-values ) ---> next-internal-state
_ffunc_( internal-state ) ---> aggregate-value
```

PostgreSQL创建一个类型为 `_stype_` 的临时变量。 它保存这个聚集的当前内部状态。对于每个输入数据条目，都调用状态转换函数计算内部状态值的新数值。 在处理完所有数据后，调用一次最终处理函数以计算聚集的返回值。如果没有最终处理函数， 则将最后的状态值当做返回值。

一个聚集函数还可能提供一个初始条件，也就是内部状态值的初始值。 这个值是作为一个类型为 `text` 的字段存储在数据库里的，不过它们必须是状态值数据类型的合法的外部 表现形式的常量。 如果没有提供状态，那么状态值初始化为 NULL 。

如果该状态转换函数被定义为"strict"，那么就不能用 NULL 输入调用它。 此时， 聚集的执行如下所述。带有任何 NULL 输入值的行将被忽略(不调用此函数并且保留前一个状态值)。 如果初始状态值是 NULL ，那么在第一个含有非 NULL 值的行上，使用第一个参数值替换状态值， 然后状态转换函数在随后所有的含有非 NULL 值的行上调用。 这样做让比较容易实现像 `max` 这样的聚集。 请注意这种行为只是当 `_state_data_type_` 与 `_input_data_type_` 相同的时候才表现出来。 如果这些类型不同，你必须提供一个非 NULL 的初始条件或者使用一个非"strict"的状态转换函数。

如果状态转换函数不是严格(strict)的，那么它将无条件地在每个输入行上调用，并且必须自行处理 NULL 输入和NULL 转换值，这样就允许聚集的作者对聚集中的 NULL 有完全的控制。

如果最终转换函数定义为"strict"，那么如果最终状态值是NULL 时就不会调用它； 而是自动输出一个 NULL 结果。(这才是 strict 函数的正常特征。) 不管是那种情况，最终处理函数 可以自由选择是否返回 NULL 。 比如， `avg` 的最终处理函数在零输入记录时 就会返回 NULL 。

行为类似 `MIN` 或 `MAX` 的聚集有时候可以优化为使用索引，而不用扫描 每个输入行。 如果这个聚集可以如此优化，则用一个排序操作符标识它。 这里基本的要求是聚集 必须以操作符归纳出来的排序顺序生成第一个元素；换句话说：

```
SELECT agg(col) FROM tab;
```

必须等于：

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

更多的假设是聚集忽略 NULL 输入，并且只有在输入没有非空数值的时候，它才生成NULL结果。 通常，数据类型的 `<` 操作符是 `MIN` 的适用排序操作符，而 `>` 是 `MAX` 的适用操作符。 请注意，除非声明的操作符是 B-tree 索引操作符类的"小于"或者"大于"策略号， 否则这种优化将不会生效。

要创建聚集函数，你必须具有相关参数类型、状态类型和返回类型的 `使用` 权限，也必须具有转换函数和最终函数的 `执行` 权限。

参数

`_name_`

要创建的聚集函数名(可以有模式修饰) 。

`_input_data_type_`

该聚集函数要处理的输入数据类型。要创建一个零参数聚集函数，可以使用 `*` 代替输入数据类型列表。 （ `count(*)` 就是这种聚集函数的一个实例。 ）

`_base_type_`

在以前的 `CREATE AGGREGATE` 语法中，输入数据类型是通过 `basetype` 参数指定的，而不是写在聚集的名称之后。 需要注意的是这种以前语法仅允许一个输入参数。 要创建一个零参数聚集函数，可以将 `basetype` 指定为 `"ANY"` (而不是 `*`)。

`_sfunc_`

将在每一个输入行上调用的状态转换函数的名称。 对于有 `_N_` 个参数的聚集函数， `_sfunc_` 必须有 `_+1_` 个参数，其中的第一个参数类型为 `_state_data_type_` ，其余的匹配已声明的输入数据类型。 函数必须返回一个 `_state_data_type_` 类型的值。 这个函数接受当前状态值和当前输入数据，并返回下个状态值。

`_state_data_type_`

聚集的状态值的数据类型。

`_ffunc_`

在转换完所有输入行后调用的最终处理函数，它计算聚集的结果。 此函数必须接受一个类型为 `_state_data_type_` 的参数。 聚集的输出数据 类型被定义为此函数的返回类型。 如果没有声明 `_ffunc_` 则使用聚集结果的状态值作为聚集的结果，且输出类型为 `_state_data_type_` 。

`_initial_condition_`

状态值的初始设置(值)。 它必须是一个 `_state_data_type_` 类型可以接受的文本常量值。 如果没有声明，状态值初始为 NULL 。

`_sort_operator_`

用于 `MIN` 或 `MAX` 类型聚集的排序操作符。 这个只是一个操作符名 (可以有模式修饰)。这个操作符假设接受和聚集一样的输入数据类型。

`CREATE AGGREGATE` 的参数可以以任何顺序书写，而不只是上面显示的顺序。

例子

参见Section 35.10。

兼容性

`CREATE AGGREGATE` 是PostgreSQL语言的扩展。 SQL标准没有提供用户自定义聚集函数的功能。

参见

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

Name

CREATE CAST -- 定义一个用户定义的转换

Synopsis

```
CREATE CAST (_source_type_ AS _target_type_)
    WITH FUNCTION _function_name_ (_argument_type_ [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (_source_type_ AS _target_type_)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (_source_type_ AS _target_type_)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

描述

`CREATE CAST` 定义一个新的转换。一个转换说明如何在两个类型之间进行转换。比如：

```
SELECT CAST(42 AS float8);
```

通过调用前面指定的函数将整数常量42转化为 `float8` 类型，即 `float8(int4)` 的形式。 （如果没有定义合适的转换，转换将失败。）

两种形式是二进制可强制转换的，这意味转换可以在不调用函数的情况下自由执行。 这要求相应的值使用相同的内部表示。例如， `text` 和 `varchar` 形式都是二进制可强制转换的两种类型。 二进制可可强制转换未必是一个对称的关系。例如： `xml` 到text的转换可以在当前的处理中可直接执行。 但相反的方向的转换需要一个函数来执行，至少一个语法检查。（两种二进制可强制转换的形式也被称为二进制兼容。）

您可以定义一个转换为使用 `WITH INOUT` 语法的*I/O conversion cast*转化转换。 一个I/O转化转换通过调用原数据库类型的输出函数来执行，并将结果传给目标数据类型的输入函数。

缺省时，只有在明确要求转换的情况下才调用一个转换，也就是一个明确的 `CAST(``_x_` AS `_typename_`)或 `_x_``::``_typename_` 转换要求。

如果转换被标记为 `AS ASSIGNMENT` ，那么在给目标数据类型的字段赋值的时候，可以隐含调用它。 比如，假设 `foo.f1` 是一个类型为 `text` 的字段， 那么：

```
INSERT INTO foo (f1) VALUES (42);
```

如果从 `integer` 类型到 `text` 类型的转换标记为 `AS ASSIGNMENT`，上面的这句就被允许，否则就不允许。（通常用术语*assignment cast*来描述这种转换。）

如果转换标记了 `AS IMPLICIT`，那么可以在任何环境下调用，不管是作业还是在一个内部表达式中。（我们通常使用术语*implicit cast*来描述这种转换。）例如，考虑下面这个查询：

```
SELECT 2 + 4.0;
```

解析器初始时标记常量分别为 `integer` 和 `numeric`。在系统目录中没有 `integer` + `numeric` 的操作符，但是有一个 `integer` + `numeric` 操作符。若 `integer` 到 `numeric` 的转换是可以执行的并且标记为 `AS IMPLICIT` — 则该查询将会成功执行。解析器将使用隐性的转换并按所写的查询生成结果：

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

现在，目录提供了一个从 `numeric` 到 `integer` 的转换。如果那个转换标记了 `AS IMPLICIT` — 而它并不是— 然后解析器会面临对选择上面的解释还是选择 `numeric` 常量到 `integer` 的转换，然后应用 `integer` `+` `integer` 操作符。缺乏选择哪一种处理方式的足够信息，系统会放弃执行并返回查询是模棱两可的信息。两种转换中仅有一个是缺省的方式正是我们设计的方式，我们让解析器更偏向于将 `numeric` -和- `integer` 的混合表达式的结果视作 `numeric`；没有关于那方面的内置信息。

在是否将转换标记为隐性的问题上保守一些是明智的。过于丰富的隐含转换路径 会导致PostgreSQL选择让人奇怪的命令解析，或者是完全不能解析命令，因为存在多个可能的解析。一个好的规则是，只有在同一个通用类型范畴里面的那些可以保留转换信息的类型之间才标记为可隐含调用转换。比如，从 `int2` 到 `int4` 可以合理地标记为隐含转换，但是从 `float8` 到 `int4` 也许应该标记为赋值转换。跨类型范围的转换，比如 `text` 到 `int4`，只能明确地转换。

> **Note:** 有时，有必要为了可用性和标准支持的原因在一组类型中提供多个隐含转换，导致上述无法避免的模棱两可的问题。解析器有一个基于*type categories*和*preferred types*的启发式函数回调功能，有助于在这种情况下提供所期望的行为。参阅CREATE TYPE获取更多详细信息。

为了能够创建一个转换，您必须是源或者目标数据类型的所有者。为了创建一个强制二进制的转换，您必须是超级用户。（做这个约束的原因是错误的二进制可强迫转换转换可以很容易让服务器崩溃。）

参数

`_source_type_`

转换的源数据类型。

`_target_type_`

转换的目标数据类型。

`_function_name_` (`_argument_type_` [, ...])

用于执行转换的函数。 这个函数名可以是用模式名修饰的。 如果它没有用模式名修饰， 那么该函数将从模式搜索路径中找出来。 函数的结果数据类型必须匹配转换的目标类型。 它的参数在下面讨论。

`WITHOUT FUNCTION`

表明源类型是对目标类型是二进制可强制转换的，所以没有函数需要执行此转换。

`WITH INOUT`

表明转换是I/O转换，通过调用源数据类型的输出函数来执行，并将结果传给目标数据类型的输入函数。

`AS ASSIGNMENT`

表示转换可以在赋值模式下隐含调用。

`AS IMPLICIT`

表示转换可以在任何环境里隐含调用。

转换实现函数可以有一到三个参数。 第一个参数的类型必须与转换的源类型相同的，或可以从转换的源类型二进制可强制转换的。 第二个参数，如果存在，必须是 `integer` 类型；它接收这些与目标类型相关联的类型修饰符，或者若什么都没有则是-1。 第三个参数，如果存在，必须是 `boolean` 类型；若转换是一个显式类型转换则会收到 `true` ， 否则是 `false` 。 （奇怪的是，在一些情况下SQL标准要求对显式和隐式转换的不同表现。我们不推荐您设计自己的数据类型，这很重要。）

一个转换函数的返回类型必须是与转换的目标类型相同或者对转换的目标类型二进制可强制转换 。

通常，一个转换必须有不同的源和目标数据类型。 然而，若有多于一个参数的转换实现函数，则允许声明一个有相同的源和目标类型的转换。 这用于表示系统目录中的特定类型的长度强制函数。 命名的函数用于强制一个该类型的值为第二个参数给出的类型修饰符值。

如果一个类型转换的源类型和目标类型不同，并且接收多于一个参数，它就表示从一种类型转换成另外一种类型只用一个步骤，并且同时实施长度转换。 如果没有这样的项可用， 那么转换成一个使用了类型修饰词的类型将涉及两个步骤，一个是在数据类型之间转换， 另外一个是施加修饰词指定的转换。

对域类型的转换目前没有作用。转换一般是针对域相关的所属数据类型。

注意

使用DROP CAST删除用户定义的转换。

请注意，如果希望能双向转换类型，那么你需要明确地定义两个方向的转换。

通常不需要创建用户定义类型与标准字符串类型之间的转换 (`text` ， `varchar` 和 `char(``_n_`)，以及被定义为字符串的范畴的用户定义的类型)。PostgreSQL为此提供自动I/O转换转换。 字符串类型的自动转换可以认为是分配转换，而来自字符串的自动转换是唯一显式的。 您可以通过声明自己的转换替换系统的自动转换，但是，通常这么做的唯一原因是，你想让转化比标准唯一分配或者唯一显式设置更容易调用。 另一个可能的原因是你想让转化变现的不同于类型的I/O函数； 但是最重要的是您应该反复考虑这是否是一个好主意。（少量内部类型确实对转换有不同的性能要求，大部分是因为要求SQL标准。）

在PostgreSQL7.3之前，每个与数据类型名称相同的函数会返回那个数据类型，并取一个不同类型的参数的函数，自动成为一个转换函数。 在面临模式引入时约定已取消并且能代表系统表中的二进制可强制转换。内置的转换功能仍然遵循这种命名模式，但是他们必须像系统表 `pg_cast` 中的转换一样显示。

虽然不是必须的，但是还是建议你遵循旧的命名类型转换实现函数的习惯，也就是说， 函数名和目标数据类型同名。 许多用户习惯于使用函数风格的表示法 `_typename_` (`_x_`)来做数据类型转换。 这种表示法恰好就是调用类型转换实现函数，这样并不会被当作一种类型转换而被特殊看待。 如果你的转换函数没有按照这种传统命名，那么你就会让用户很奇怪。 因为PostgreSQL允许同名不同参数的函数重载，因此同时存在多个从不同类型向同样类型转换的同名转换函数一点问题都没有。

> **Note:** 事实上前面所述是过分简化的：在两种情况下函数调用结构被认为是一个转换请求而不需要将其匹配为一个实际函数。 如果函数调用 `_name_` (`_x_`)不准确匹配任何现有函数，但是 `_name_` 是一个数据类型的名称 并且 `pg_cast` 从 `_x_` 类型提供了一个二进制可强制转换到这个类型，则调用会被解析为一个二进制可强制转换。 即使是没有任何转换函数，也设计了一种异常，二进制可强制转换可以通过使用函数语法来调用， 同样的，若无 `pg_cast` 条目，但转换会到达或者来自一个字符串类型， 调用将会被视为一个I/O转换转换。 该异常情况下允许I/O转换转换通过使用函数语法来调用。
>
> **Note:** 也还有一种异常中的异常：从复合数据类型向字符串类型的I/O转换不能使用函数语法，必须设计为精确的转换语法（ `CAST` 或 `::` 声明） 增加这种异常是因为在介绍过自动执行的I/O转换后，用户会发现在一个函数或是字段参考时很容易误执行一个类似的转换。

例子

为了从类型 `bigint` 到类型 `int4` 创建一个指派映射要通过使用函数 `int4(bigint)`：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（这个转换在系统中已经预先定义了。）

兼容性

`CREATE CAST` 指令符合SQL标准，除了SQL没有为二进制可强制转换类型或者实现函数的额外参数来实现功能。 `AS IMPLICIT` 也是一个PostgreSQL扩展。

参见

CREATE FUNCTION、CREATE TYPE 和DROP CAST

CREATE COLLATION

Name

CREATE COLLATION -- 定义一个新的排序规则

Synopsis

```
CREATE COLLATION _name_ (
    [ LOCALE = _locale_, ]
    [ LC_COLLATE = _lc_collate_, ]
    [ LC_CTYPE = _lc_ctype_ ]
)
CREATE COLLATION _name_ FROM _existing_collation_
```

描述

`CREATE COLLATION` 使用指定操作系统中的编码设置定义一个新的排序规则，或是从一个已存在的排序规则中进行复制。

要创建一个排序规则，你必须在目标模式中有 `CREATE` 权限。

参数

`_name_`

排序规则的名称。排序规则的名称可以有模式修饰符。如果没有，则排序规则是定义在当前模式中。 另外排序规则在指定的模式中名称必须是唯一的。 （系统目录中允许含有不同内部字符编码的相同排序规则名，如果是数据库的编码不相匹配，则它们会被忽略。）

`_locale_`

这是一次性设置 `LC_COLLATE` 和 `LC_CTYPE` 的快捷方式。 如果你定义了这个变量，就不能定义其他两个变量了。

`_lc_collate_`

使用指定操作系统中的字符编码来设置 `LC_COLLATE` 。 字符编码对当前数据库来说必须是可使用的。 （参见CREATE DATABASE了解更精确的信息。）

`_lc_ctype_`

使用指定操作系统中的字符编码来设置 `LC_CTYPE` 。 字符编码对当前数据库来说必须是可使用的。（参见CREATE DATABASE了解更精确的信息。）

`_existing_collation_`

复制已存在的排序规则。新的排序规则将和已存在的规则有相同的属生，但它是一个独立的对像。

注意

使用 `DROP COLLATION` 命令可以删除一个用户定义的排序规则。

参见Section 22.2了解更多在PostgreSQL中支持的排序规则。

例子

根据操作系统中的系统字符编码 `fr_FR.utf8` 来创建一个排序规则（假定当前数据库的编码是 `UTF8`):

```
CREATE COLLATION french (LOCALE = 'fr_FR.utf8');
```

从一个已存在的排序规则中复制一个新的排序规则：

```
CREATE COLLATION german FROM "de_DE";
```

这对在应用程序中使用独立于操作系统的排序规则名是很方便的。

兼容性

在SQL标准中有 `CREATE COLLATION` 命令，但它仅限于从已存在的排序规则中拷贝。 允许创建一个新的排序规则是PostgreSQL的扩展。

参见

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

Name

CREATE CONVERSION -- 定义一个新的编码转换

Synopsis

```
CREATE [ DEFAULT ] CONVERSION _name_
    FOR _source_encoding_ TO _dest_encoding_ FROM _function_name_
```

描述

`CREATE CONVERSION` 定义字符集之间的转换。 标记为 `DEFAULT` 的转换可以用于在前端和后端之间的自动编码转换。 出于这个原因，一般的转换如从编码 A 到 B和从编码B到A，必须定义两种转换。

为了可以创建转换，你必须有函数的 `EXECUTE` 权限并且在目标模式上有 `CREATE` 权限。

参数

`DEFAULT`

`DEFAULT` 选项表示这种转换是从这种源编码到目标编码的缺省转换。 同一个模式里每一对编码应该只有一个缺省编码转换。

`_name_`

转换的名字。转换名可以用模式修饰。如果没有，那么转换就在当前模式中定义。 转换名在一个模式里必须唯一。

`_source_encoding_`

源编码名。

`_dest_encoding_`

目标编码名 。

`_function_name_`

用于执行转换的函数。这个函数名可以用模式名修饰。如果没有，那么将从路径中找出这个函数。

此函数必须有如下的样子：

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer   -- source string length
) RETURNS void;
```

注意

用 `DROP CONVERSION` 删除用户定义的转换。

创建转换所需要的权限可能在未来的版本中改变。

例子

用 `myfunc` 创建一个从 `UTF8` 编码到 `LATIN1` 编码的转换：

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

兼容性

`CREATE CONVERSION` 是PostgreSQL的扩展。 在SQL标准里没有 `CREATE CONVERSION` 语句。 但有一个在语法和意义上都很相似的 `CREATE TRANSLATION` 语句。

参见

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

Name

CREATE DATABASE -- 创建一个新数据库

Synopsis

```
CREATE DATABASE _name_
    [ [ WITH ] [ OWNER [=] _user_name_ ]
           [ TEMPLATE [=] _template_ ]
           [ ENCODING [=] _encoding_ ]
           [ LC_COLLATE [=] _lc_collate_ ]
           [ LC_CTYPE [=] _lc_ctype_ ]
           [ TABLESPACE [=] _tablespace_name_ ]
           [ CONNECTION LIMIT [=] _connlimit_ ] ]
```

描述

`CREATE DATABASE` 创建一个新PostgreSQL数据库。

要创建一个数据库，你必须是一个超级用户或者有特殊的 `CREATEDB` 权限。 参阅CREATE USER。

缺省情况下新数据库将通过复制标准系统数据库 `template1` 来创建。 可以通过 `TEMPLATE _name_` 指定不同的模板。 尤其是，用 `TEMPLATE template0` 创建一个很纯净的、 只包括PostgreSQL预定义的标准对象的数据库。 这个方法对于避免把任何已经加入到 `template1` 里的本地安装对象拷贝到新数据库是非常有用的。

参数

`_name_`

要创建的数据库名字。

`_user_name_`

数据库用户的名字，他将是新数据库的所有者，或者是使用 `DEFAULT` 选项来指定当前缺省用户(也就是执行命令的用户)。 要创建一个其他角色所有的数据库，你必须是那个角色的直接或间接的成员， 或者是超级用户。

`_template_`

模板名，即从哪个模板创建新数据库，或者使用 `DEFAULT` 选项来指定缺省模板(`template1`)。

`_encoding_`

创建新数据库使用的字符编码。 可以使用文本名字(例如 `'SQL_ASCII'`)、整数编号或是 `DEFAULT` 选项来指定(模版数据库的编码)。 PostgreSQL服务器支持的字符集在Section 22.3.1里有描述。 额外的限制参见下文。

`_lc_collate_`

用于新数据库的排序规则(LC_COLLATE)。 这影响到应用对字符串的排序顺序，例如：在有ORDER BY的查询中，以及用于文本列的索引的顺序。 在默认情况下，使用模板数据库的排序规则。 请看以下附加的限制。

`_lc_ctype_`

用于新数据库的字符分类(`LC_CTYPE`)。 这影响字符的分类，例如： 小写、大写和数字。 默认情况下使用模板数据库的字符分类。 请看以下附加的限制。

`_tablespace_name_`

和新数据库关联的表空间名字， 或者使用 `DEFAULT` 选项表示使用模版数据库的表空间。 这个表空间将成为在这个数据库里创建的对象的缺省表空间。 参阅CREATE TABLESPACE获取更多信息。

`_connlimit_`

数据库可以接受多少并发的连接。-1(缺省)意味着没有限制。

可选参数可以按任意顺序书写，而不仅仅是上面显示的顺序。

注意

`CREATE DATABASE` 不能在事务块里面执行。

类似"could not initialize database directory"这样的错误，最有可能是因为数据目录的权限不够或者磁盘满之类的文件系统问题。

使用DROP DATABASE删除一个数据库。

程序createdb是这个命令的封装，使用更加方便。

尽管可以通过把某数据库名声明为模板而不是用 `template1` ，但是这(还)不是一个通用的" `COPY DATABASE` "功能。 主要的限制是在从模版复制的时候不允许有其它会话链接到模版数据库上。 如果在开始执行 `CREATE DATABASE` 的时候有其它会话正连接在模版数据库上，那么操作将会失败； 否则直到 `CREATE DATABASE` 完成之后，才允许对模板数据库建立新的会话连接。 参见Section 21.3获取更多信息。

为一个新数据库指定的字符集编码必须兼容所选择的区域环境设置 (`LC_COLLATE` 和 `LC_CTYPE`)。 若区域设置为 `C` (或相当于 `POSIX`)，那么所有的编码都允许，但是对于其他的区域设置，只有一个编码能正常工作。 （然而，在Windows系统中， UTF-8编码可用于任何区域设置。） `CREATE DATABASE` 将会允许超级用户 来指定 `SQL_ASCII` 编码而不考虑区域设置情况，这种做法已过时了，是不宜采用的，它会导致编码与不兼容区域设置的数据被存储在数据库中时字符串函数功能会不正常。

除了 `template0` 用作模板的时候，其他数据库的编码和区域设置必须匹配模板数据库。 这是因为其他数据库可能会包含不匹配指定编码的数据，或者可能包含排序顺序受 `LC_COLLATE` 和 `LC_CTYPE` 影响的索引。 复制这些数据会导致数据库被新设置破坏。 然而， `template0` 公认是不包含任何会受到影响的数据或者索引的。

`CONNECTION LIMIT` 选项只是近似地强制；如果两个新的连接几乎同时发起， 而只剩下一个连接"slot"了，那么很可能两个连接都失效。 另外，超级用户连接时不受这个限制。

例子

创建一个新数据库:

```
CREATE DATABASE lusiadas;
```

创建一个由用户 `salesapp` 拥有的数据库 `sales` ，缺省表空间是 `salesspace` ：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

创建一个数据库 `music` ，支持ISO-8859-1字符集：

```
CREATE DATABASE music ENCODING 'LATIN1' TEMPLATE template0;
```

在这个例子中， `TEMPLATE template0` 选项只在 `template1` 的编码不是ISO-8859-1时被请求。 请注意：更改编码可能也会需要选择新的 `LC_COLLATE` 和 `LC_CTYPE` 设置。

兼容性

在sql标准里头没有 CREATE DATABASE 语句。数据库等同于目录，其创建是交给具体的数据库实现去定义的。

参见

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

Name

CREATE DOMAIN -- 定义一个新域

Synopsis

```
CREATE DOMAIN _name_ [ AS ] _data_type_
    [ COLLATE _collation_ ]
    [ DEFAULT _expression_ ]
    [ _constraint_ [ ... ] ]

where `_constraint_` is:

[ CONSTRAINT _constraint_name_ ]
{ NOT NULL | NULL | CHECK (_expression_) }
```

描述

`CREATE DOMAIN`创建一个新的数据域。 域在本质上是一个带有可选约束的数据类型(限制允许的取值范围)。定义域的用户成为其所有者。

如果给出一个模式名称(比如`CREATE DOMAIN myschema.mydomain ...`)， 那么该域是在指定的模式中创建的。 否则它会在当前模式中创建。域名字必需在其所在模式中的现有类型和域中唯一。

域可以便于把不同表之间的公共约束提取到一个固定位置进行维护。 比如，在多个表中都含有一个电子邮件地址字段，这个字段都要求有CHECK的约束检查来验证邮箱的有效性。 此时可以定义并使用一个域，而不是分别设置每个表的约束。

要创建域，你必须有相关数据类型的`USAGE`权限。

参数

`_name_`

要创建的域名字(可以有模式修饰)。

`_data_type_`

域的下层数据类型。这时可以包含一组的定义选项。

`_collation_`

可选的域的排序规则。如果没有指定排序规则，会使用下层所属数据类型的排序规则。 如果使用了 `COLLATE` 选项，则下层数据类型必须是可排序的。

`DEFAULT` `_expression_`

`DEFAULT` 选项为域数据类型的字段声明一个缺省值。 该值是任何不含变量的表达式(但不允许子查询)。 缺省表达式的数据类型必需匹配域的数据类型。 如果没有声明缺省值，那么缺省值就是 NULL 。

缺省表达式将用于任何省略该字段值的插入操作。 如果为特定的字段声明了缺省值， 那么它覆盖任何和该域相关联的缺省值。 然后，域的缺省覆盖任何与下层数据类型相关的缺省。

`CONSTRAINT` `_constraint_name_`

一个约束的可选名称。如果没有声明，系统将自动生成一个名字。

`NOT NULL`

域的值通常不能为空。 然而，对于有此约束的域，若分配了匹配的空域类型，则仍然可能会有空值， 例如：通过一个LEFT OUTER JOIN或者 `INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false))` 。

`NULL`

这个域的数值允许为 NULL 。这是缺省。

这个子句只是用于和非标准的 SQL 数据库兼容用。不建议在新的应用中使用它。

`CHECK (``_expression_`)

`CHECK` 选项声明域的数值必须满足的完整性约束或测试。 每个约束必须是一个生成布尔结果的表达式。它应该使用关键字 `VALUE` 来引用被测试的数值。

目前， `CHECK` 表达式不能包含子查询，也不能引用除 VALUE之外的变量。

例子

这个例子创建了 `us_postal_code` 数据类型并且在一个表定义中 使用了该类型。 使用了一个正则表达式测试以保证这些数值看起来像一个美国的邮政编码：

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

兼容性

`CREATE DOMAIN` 命令符合SQL标准。

参见

ALTER DOMAIN, DROP DOMAIN

CREATE EXTENSION

Name

CREATE EXTENSION -- 安装一个扩展

Synopsis

```
CREATE EXTENSION [ IF NOT EXISTS ] _extension_name_
    [ WITH ] [ SCHEMA _schema_name_ ]
             [ VERSION _version_ ]
             [ FROM _old_version_ ]
```

描述

create extension 命令安装一个新的扩展到一个数据库中.必须保证没有同名的扩展已经被安装.

安装一个扩展意味着执行一个扩展的脚本文件.这个脚本会创建一个新的SQL实体,例如函数,数据类型,操作符,和索引支持的方法.

安装扩展需要有和创建他的组件对象相同的权限.对于大多数扩展这意味着需要超户或者数据库所有者的权限. 对于后续的权限检查和该扩展脚本所创建的实体，运行CREARE EXTENSION命令的角色将变为扩展的所有者.

参数

`IF NOT EXISTS`

如果系统已经存在一个同名的扩展，不会报错．这种情况下会给出一个提示．请注意该参数不保证系统存在的扩展和现在脚本创建的扩展相同.

`_extension_name_`

将被安装扩展的名字．PostgresQL 从 `SHAREDIR/extension/``_extension_name_``.control` 这个文件安装扩展.

`_schema_name_`

扩展的实例被安装在该模式下,扩展的内容可以被重新安装.指定的模式必须已经存在.如果没有指定,扩展的控制文件也不指定一个模式,这样将使用默认模式.

注意扩展不认为它在任何模式里面:扩展在一个数据库范围内的名字是不受限制的,但是一个扩展的实例是属于一个模式的.

`_version_`

安装扩展的版本.这个可以写为一个标识符或者字符串.默认的版本在扩展的控制文件中指定.

`_old_version_`

当你想升级安装"old style" 模块中没有的内容时,你必须指定 `FROM` `_old_version_`. 这个选项使 `CREATE EXTENSION` 运行一个安装脚本将新的内容安装到扩展中,而不是创建一个新的实体.注意 `SCHEMA` 指定了包括这些已存在实体的模式.

`_old_version_` 的值由扩展的作者决定,如果有多个旧的版本升级到一个扩展这个值可能会改变.对于pre-9.1PostgreSQL 提供的标准扩展,当升级扩展时使用 `unpackaged` 代替 `_old_version_`

注意

在你使用 `CREATE EXTENSION` 去安装一个扩展到数据库之前,扩展所需的文件必须被安装.安装扩展所需的信息在Additional Supplied Modules.

现在可以安装的扩展可以在 `pg_available_extensions` 或者 `pg_available_extension_versions` 找到.

关于写一个扩展信息,可以查看Section 35.15.

例子

在当前数据库安装hstore扩展:

```
CREATE EXTENSION hstore;
```

升级一个9.1安装的 `hstore` :

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

注意指定你在哪个模式安装过 `hstore` .

兼容性

`CREATE EXTENSION` 是PostgreSQL的一个扩展.

相关内容

ALTER EXTENSION, DROP EXTENSION

CREATE EVENT TRIGGER

Name

CREATE EVENT TRIGGER -- 定义一个事件触发器

Synopsis

```
CREATE EVENT TRIGGER _name_
  ON _event_
  [ WHEN _filter_variable_ IN (filter_value [, ... ]) [ AND ... ] ]
  EXECUTE PROCEDURE _function_name_()
```

描述

`CREATE EVENT TRIGGER` 创建一个新的事件触发器.无论何时指定的事件发生或 `WHEN` 的条件满足触发器，触发器函数将被执行. 对于事件触发器的一般说明，请见于Chapter 37. 创建事件触发器的用户将变为它的拥有者.

参数

`_name_`

一个新的触发器的名字. 这个名字必须在数据库内是唯一的.

`_event_`

触发调用一个给定函数的事件名子.关于事件名字的更多信息见于 Section 37.1.

`_filter_variable_`

过滤事件的变量名称.这将限制它所支持的事件的一个子集去触发该触发器.现在仅支持 `_filter_variable_` 值为TAG.

`_filter_value_`

可以触发该触发器的 `_filter_variable_` 相关的值.对于 `TAG` ,这意味着一个tags的命令列表.(例如. `'DROP FUNCTION'`).

`_function_name_`

一个用户声明的不带参数的函数并且返回 `event_trigger` 类型.

Notes

只有超户能创建事件触发器.

在单用户模式下事件触发器是被关闭的(详见于postgres). 如果一个错误的事件触发器关闭了数据库,在单用户模式下你将不能重启数据库,删除事件触发器.

Examples

禁止执行任何DDL命令:

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE PROCEDURE abort_any_command();
```

Compatibility

在标准的SQL语法中没有 `CREATE EVENT TRIGGER` 语句.

相关内容

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE FOREIGN DATA WRAPPER

Name

CREATE FOREIGN DATA WRAPPER -- 定义一个外部数据封装器

Synopsis

```
CREATE FOREIGN DATA WRAPPER _name_
    [ HANDLER _handler_function_ | NO HANDLER ]
    [ VALIDATOR _validator_function_ | NO VALIDATOR ]
    [ OPTIONS ( _option_ '_value_' [, ... ] ) ]
```

描述

`CREATE FOREIGN DATA WRAPPER` 创建一个新的外部数据封装器。 创建外部数据封装器的用户成为其所有者。

外部数据封装器的名字必需在数据库中唯一。

只有超级用户可以创建外部数据封装器。

参数

`_name_`

要创建的外部数据封装器的名字。

`HANDLER` `_handler_function_`

`_handler_function_` 是先前已经注册了的函数的名字，用来为外部表检索执行函数。 处理器函数必须没有参数，并且它的返回类型必须为 `fdw_handler` 。

不用处理器函数创建外部数据封装器是可能的，但是使用这种封装器的外部表只能被声明，不能被访问。

`VALIDATOR` `_validator_function_`

`_validator_function_` 是先前已经注册了的函数的名字，用来检查提供给外部数据封装器的通用选项， 还有使用该外部数据封装器的外部服务器和用户映射的选项。 如果没有验证器函数或声明了 `NO VALIDATOR` ， 那么在创建时将不检查选项。（外部数据封装器可能在运行时忽

略或拒绝无效的选项说明， 取决于实现。）验证器函数必须接受两个参数：一个类型为 `text[]` ， 将包含存储在系统目录中的选项的数组，一个类型为 `oid` ， 是包含这些选项的系统目录的OID。忽略返回类型；该函数应该使用 `ereport(ERROR)` 函数报告无效选项。

`OPTIONS (` `_option_` ' `_value_` '[, ...])

这个子句为新的外部数据封装器声明选项。允许的选项名和值是特定于每个外部数据封装器的， 并且是经过外部数据封装器的验证器函数验证了的。选项名必须是唯一的。

注意

PostgreSQL的外部数据功能一直在积极开发。 查询的优化是原始的（并且主要是封装器）。因此，未来的性能提升有很大的空间。

例子

创建一个无用的外部数据封装器 `dummy` ：

```
CREATE FOREIGN DATA WRAPPER dummy;
```

创建一个带有处理器函数 `file_fdw_handler` 的外部数据封装器 `file` ：

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

创建一个带有一些选项的外部数据封装器 `mywrapper` ：

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

兼容性

`CREATE FOREIGN DATA WRAPPER` 遵从ISO/IEC 9075-9 (SQL/MED)， 除了 `HANDLER` 和 `VALIDATOR` 子句是扩展， 并且标准的子句 `LIBRARY` 和 `LANGUAGE` 没有在PostgreSQL中实现。

请注意，然而，SQL/MED功能作为一个整体目前是不符合的。

又见

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

Name

CREATE FOREIGN TABLE -- 定义一个新外部表

Synopsis

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] _table_name_ ( [
    _column_name_ _data_type_ [ OPTIONS ( _option_ '_value_' [, ... ] ) ] [ COLLATE _coll
    [, ... ]
] )
  SERVER _server_name_
[ OPTIONS ( _option_ '_value_' [, ... ] ) ]
这里`_column_constraint_` 可以是:

[ CONSTRAINT _constraint_name_ ]
{ NOT NULL |
  NULL |
  DEFAULT _default_expr_ }
```

描述

`CREATE FOREIGN TABLE` 在当前数据库中创建一个外部表，该表将由发出此命令的用户所有。

如果给出了模式名 (for example, `CREATE FOREIGN TABLE myschema.mytable ...`)，那么表是在指定模式中创建的。否则在当前模式中创建外部表。 外部表的名字必须与同一个模式中的其它外部表，表，序列，索引或试图的名字不同

`CREATE FOREIGN TABLE` 还自动创建一个数据类型，该数据类型代表对应该外部表一行的复合类型。 因此，外部表不能和同模式中的现有数据类型同名。

为了创建一个外部表，除了对外部表所有字段类型有 `USAGE` 权限外， 还必须有外部表服务器的 `USAGE` 权限。

参数

`IF NOT EXISTS`

如果已经存在相同名称的对象，在这种情况下，不会抛出错误，只会产生一个通知。 请注意这并不保证将要创建的对象与现有对象是否一致。

`_table_name_`

要创建的表的名字（可以用模式修饰）。

`_column_name_`

新表中要创建的字段名。

`_data_type_`

该字段的数据类型. 它可以包含数组说明符。有关 PostgreSQL 支持的数据类型的更多信息，请参考Chapter 8。

`NOT NULL`

该字段不允许包含null值。

`NULL`

该字段允许包含null值。 这是缺省。

这个子句的存在只是为和那些非标准 SQL 数据库兼容。 我们不建议在新应用中使用它。

`DEFAULT` `_default_expr_`

`DEFAULT` 子句给它所出现的字段设定一个缺省数值。该数值可以是任何不含变量的表达式（不允许使用子查询和对本表中的其它字段的交叉引用）。缺省表达式的数据类型必须和字段类型匹配。

缺省表达式将被用于任何未指定该字段数值的插入操作。 如果字段上没有缺省值，那么缺省是 NULL。

`_server_name_`

外部表使用的已存在的外部服务器名称。更多细节，参考CREATE SERVER。

`OPTIONS (` `_option_` ' `_value_` ' [, ...])

选项与新外部表或外部表中的字段有关。允许的选项名称和值，是由每一个外部数据封装器中来说是特别指定的。 也是通过外部数据封装器的验证函数来验证。重复的选项名称是不被允许的(尽管表选项和表字段选项可以有相同的名字)。

例子

创建外部表 `films`，该表通过服务器 `film_server` 访问：

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

兼容性

`CREATE FOREIGN TABLE` 命令最大程度上符合了SQL标准； 然而， 就像使用 `CREATE TABLE` , `NULL` 约束和 零字段外部表以及设定默认值的功能是PostgreSQL对SQL标准的扩展。

See Also

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER

CREATE FUNCTION

Name

CREATE FUNCTION -- 定义一个新函数

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
    _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [ { DEFAULT | = } _default_expr_ ] [
    [ RETURNS _rettype_
      | RETURNS TABLE ( _column_name_ _column_type_ [, ...] ) ]
  { LANGUAGE _lang_name_
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST _execution_cost_
    | ROWS _result_rows_
    | SET _configuration_parameter_ { TO _value_ | = _value_ | FROM CURRENT }
    | AS '_definition_'
    | AS '_obj_file_', '_link_symbol_'
  } ...
    [ WITH ( _attribute_ [, ...] ) ]
```

描述

`CREATE FUNCTION` 定义一个新的函数。 `CREATE OR REPLACE FUNCTION` 如果函数不存在就创建一个新函数，否则替换现有的定义。用户必须有定义该函数所用语言的 `USAGE` 权限才能定义新函数。

如果包含了一个模式名，那么函数就在指定的模式中创建。否则它会在当前模式中创建。 新函数的名字不能和同一个模式中的任何带有同样参数类型的函数同名。 不过，参数类型不同的函数可以同名(这叫做重载)。

使用 `CREATE OR REPLACE FUNCTION` 替换一个现有函数的定义。 不能用这个方法修改一个函数的名字或者参数类型， 否则就会创建一个新的函数。同样 `CREATE OR REPLACE FUNCTION` 也不会允许你修改一个现有函数的返回类型。要做这些事情， 你必须删除并重新创建函数。如果使用 `OUT` 参数， 那就意味着除了删除函数之外，你不能修改任何 `OUT` 参数的类型或者名字。

当使用 `CREATE OR REPLACE FUNCTION` 替换现有函数的定义时， 不会改变函数的属主和权限。 函数其他的属性会被赋予命令中给定的值或默认值。 只有函数的属主才可以替换函数（也可以是属主角色的成员）。

如果你删除然后重建一个函数，新函数和旧函数将是不同的实体； 你就需要删除现有引用了老函数的规则、视图、触发器等等。 使用 `CREATE OR REPLACE FUNCTION` 可以在不破坏引用该函数的对象的前提下修改函数定义。 并且，使用 `ALTER FUNCTION` 能修改一个已有函数的大多数属性。

创建这个函数的用户将成为函数的所有者。

你必须拥有要创建函数的参数类型和返回值类型的 `USAGE` 权限， 才能创建该函数。

参数

`_name_`

要创建的函数名字(可以用模式修饰)

`_argmode_`

参数的模式： `IN` ， `OUT` ， `INOUT` , 或 `VARIADIC` 。缺省值是 `IN` 。只有 `OUT` 模式的参数后面能跟 `VARIADIC` 。并且 `OUT` 和 `INOUT` 模式的参数不能用在 `RETURNS TABLE` 的函数定义中。

`_argname_`

一个参数的名字。有些语言(包括 SQL 和 PL/pgSQL)允许你在函数体里使用参数名字。 对于其它语言，输入参数名字只是额外的文档，这只是就函数定义本身而言的； 在调用函数时你可以使用输入参数名字来提高可读性。 （见 Section 4.3）。 无论如何，输出参数的名字是非常重要的，因为它定义了结果行类型的列名。 （如果你省略了输出参数的名字，那么系统会自动选择一个缺省的列名。）

`_argtype_`

该函数的数据类型(可以有模式修饰)，如果有的话。可以是基本类型， 也可以是复合类型、域类型、或者可以引用一个现有字段相同的类型。

根据实现语言的不同，还可以在这上面声明"伪类型" (比如 `cstring`)。伪类型表示实际的参数类型要么是没有完整地声明， 要么是在普通的 SQL 数据类型之外。

一个字段的类型是用 `_table_name_` . `_column_name_` %TYPE 表示的；使用这个特性有时候可以帮助创建一个不受表定义变化影响的函数。

`_default_expr_`

当参数值没有指定时作为参数默认值的表达式。该表达式的类型必须可转化为参数的类型。只有输入(也包括 `INOUT`)参数才能有默认值。具有默认值参数的输入参数必须在参数列表的最后。

`_rettype_`

返回值的数据类型。可以声明为一个基本类型、复合类型、域类型、或者引用一个表的现有字段类型。 根据实现语言的不同，还可以在这上面声明"伪类型"(比如 `cstring`)。 如果不打算返回任何值可以指定 `void` 作为返回类型。

如果存在 `OUT` 或 `INOUT` 参数，那么可以省略 `RETURNS` 子句。如果出现了， 那么它必须隐含的和输出参数结果类型兼容：如果有多个输出参数， 则必须是 `RECORD` ，如果只有一个输出参数，则与其相同。

`SETOF` 修饰词表示该函数将返回一个集合，而不是单独一条。

一个字段的类型是通过 `_table_name_` . `_column_name_` %TYPE 引用的。

`_column_name_`

`RETURNS TABLE` 的语法中输出字段名。 这是另一种有效的声名带名字的 `OUT` 类型参数的方式， 而且 `RETURNS TABLE` 隐含 `RETURNS SETOF` 。

`_column_type_`

`RETURNS TABLE` 语法中输出字段的数据类型。

`_lang_name_`

用以实现函数的语言的名字。可以是 `SQL` , `C` , `internal` ，或者是用户定义的过程语言名字。该名字可以用单引号包围。

`WINDOW`

`WINDOW` 表示该函数不是普通函数而是一个窗口函数。 这个属性当前仅对用C写的函数起作用。 当替换已有函数定义时不能改变函数的 `WINDOW` 属性。

`IMMUTABLE``STABLE` `VOLATILE`

这些属性是在查询优化器中用来优化函数的调用的，只能指定一个。缺省值是 `VOLATILE` 。

`IMMUTABLE` 表示该函数不能修改数据库， 并且在给出同样的参数值时总是返回同样的结果；也就是说，它不查询数据库或者只使用那些没有出现在参数列表里的信息。 如果给出这个选项，那么任何全部使用常数对该函数的调用都将立即替换为该函数的值。

`STABLE` 表示该函数不能修改数据库， 对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。 这个选项对那些结果依赖数据库查找、参数变量(比如当前时区)之类的函数很合适。 （但对于那些希望查询当前命令修改的行的 `AFTER` 类型的触发器是不合适的。） 还要注意 `current_timestamp` 函数族是稳定的，因为它们的值在一次事务中不会变化。

`VOLATILE` 表示该函数值甚至可以在一次表扫描内改变， 因此不会做任何优化。只有很少的数据库函数在这个概念上是易变的； 一些例子是 `random()` , `currval()` , `timeofday()` 。 请注意任何有副作用的函数都必需列为易变类，即使其结果相当有规律也应该这样， 这样才能避免它被优化；一个例子就是 `setval()` 。

更多细节，请参阅Section 35.6。

`LEAKPROOF`

`LEAKPROOF` 表示该函数没有涉密方面的副作用。 它除了返回值外，不会泄露它的参数的任何信息。例如，一个函数抛出了一个参数不正确的错误， 但错误消息里包含了参数值的信息，那这个函数就是不保密的(leakproof)。 查询计划生成器可以把保密的函数放到用 `security_barrier` 选项生成的视图中， 而不保密的函数则不可以。参见 CREATE VIEW 和 Section 38.5。 这个选项只能由超级用户设置。

`CALLED ON NULL INPUT` `RETURNS NULL ON NULL INPUT` `STRICT`

`CALLED ON NULL INPUT` (缺省)表明该函数在自己的某些参数是 NULL 的时候还是可以按照正常的方式调用。 函数的作者必须负责检查 NULL 以及进行相应地处理。

`RETURNS NULL ON NULL INPUT` 或 `STRICT` 表明如果它的任何参数是 NULL， 此函数总是返回 NULL。如果声明了这个参数，则如果存在 NULL 参数时不会执行该函数； 而只是自动假设一个 NULL 结果。

`[EXTERNAL] SECURITY INVOKER` `[EXTERNAL] SECURITY DEFINER`

`SECURITY INVOKER` (缺省)表明该函数将使用调用它的用户权限执行。 `SECURITY DEFINER` 声明该函数将以创建它的用户的权限执行。

关键字 `EXTERNAL` 的目的是和 SQL 兼容， 它是可选的，因为这个特性适用于所有函数，而不仅仅在SQL中的外部函数。 The key word is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.

`_execution_cost_`

一个正数，表示以 cpu_operator_cost 为单位的该函数的估算的执行代价。 如果函数返回的是一个结果集，它表示的是每行结果的代价。如果这个选项没有指定， 对于用C语言函数和内部函数缺省值为 1，而其他语言定义的函数缺省值为 100。 较大的值会使查询计划生成器在不必要的情况下尽量避免调用该函数。

`_result_rows_`

一个正数给出了查询计划生成器预期的该函数返回的估算的结果集行数， 它仅允许返回结果集的函数指定。缺省值为 1000。

`_configuration_parameter_` `_value_`

`SET` 语句能在进入函数时将某个配置参数设置成指定的值， 并且当函数返回时恢复成之前的值。 `SET FROM CURRENT` 使用会话当前的该参数值做为进入函数时的该参数的值。

如果一个函数创建时带有某个配置参数的 `SET` 语句， 那么在函数内对于同一参数的 `SET LOCAL` 语句的作用域仅限于该函数： 执行函数之前的该参数的值会在函数返回后恢复。然而一个一般的同一参数的 `SET` 语句（不带 `LOCAL` ）的作用效果将在函数返回后继续

保持， 直到当前的事务回滚。

关于允许设置的参数和值的进一步信息请参考 SET 和 Chapter 18。

`_definition_`

一个定义函数的字符串常量，含义取决于语言。它可以是一个内部函数名字、一个指向某个目标文件的路径、一个 SQL 查询、一个过程语言文本。

在写函数体字符串时经常使用美元符引用语法 （见Section 4.1.2.4）， 而不通常的单引号语法， 这是因为如果不使用美元符引用语法， 在函数定义中出现的任何单引号和反斜线都需要前面再加一个相应的相同字符来转义。

`_obj_file_` , `_link_symbol_`

这个形式的 `AS` 子句用于在函数的 C 源文件里的名字和 SQL 名字不同时可动态加载 C 语言函数。 字符串 `_obj_file_` 是包含可动态加载对象的文件名， 而 `_link_symbol_` 是函数的链接符号， 也就是该函数在 C 源文件里的名字。如果省略了链接符号，那么就假设它和被定义的 SQL 函数同名。

当使用同一个可动态加载文件重复执行命令 `CREATE FUNCTION` 生成函数时， 在一个全会话里该文件只会被加载一次。若想卸载或重新加载该文件（通常在开发过程中使用）， 可以开启一个新的会话。

`_attribute_`

历史遗留的函数可选信息。下面的属性可以在此出现：

`isStrict`

等效于 `STRICT` 或 `RETURNS NULL ON NULL INPUT` 。

`isCachable`

`isCachable` 是 `IMMUTABLE` 的过时的等效语法； 不过出于向下兼容，仍然接受它。

属性名是大小写无关的。

请参阅Section 35.3获取更多关于书写函数的信息。

重载

PostgreSQL 允许函数重载； 也就是只要输入参数不同，几个不同的函数可以同名。不过，所有函数的 C 名字必须不同， 也就是说你必须给予重载的 C 函数不同的 C 名字(比如，使用参数类型作为 C 名字的一部分)。

如果两个函数同名，并且输入参数类型也相同， 那么就认为这两个函数是一样的，忽略所有 `OUT` 参数。 因此，下面的声明是冲突的：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

生成同名的但有不同的参数个数的函数是没有问题的， 但是如果定义了默认值在调用时就有可能产生冲突，例如，考虑下面定义

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

函数调用 `foo(10)` 将会失败，因为系统不知道该调用哪一个函数版本。

注意

允许你将完整的 SQL 类型语法用于输入参数和返回值。 不过，有些类型声明的细节(比如 `numeric` 类型的精度域)是由下层函数实现负责的， 并且会被 `CREATE FUNCTION` 命令悄悄地吞掉(也就是不再被识别或强制)。

在使用 `CREATE OR REPLACE FUNCTION` 替换一个已存在函数的定义时， 改变函数的参数名字有一些限制。你不能对已使用了名字的输入参数改名 （但是你可以给没有使用名字的输入参数加一个名字）。如果该函数有超过一个输出参数， 你也不能改变输出参数的名字，因为改名会改变用来描述函数返回结果的匿名复合类型的列名。 这些限制都是为了使在替换函数的定义后对该函数的已有调用还能正常工作。

如果一个函数声名为 `STRICT`， 并带有一个 `VARIADIC` 模式的参数， 参数的严格性检查将可变数组整体看作非空的（non-null）。 所以当调用时参数数组有空（null）元素时， 该函数仍然会被调用。

例子

这里是一些简单的例子，用于帮助你开始掌握这个命令。更多信息和例子， 参阅Section 35.3。

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

利用参数名用 PL/pgSQL 自增一个整数：

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

返回一个包含多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

你可以通过命名明确的复合类型的方法冗长地干同样的事情：

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

另一个返回多列的方法是用 `TABLE` 函数：

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

然而， `TABLE` 是与前面的例子完全不同的， 因为它实际上返回的是一个结果集，而不仅仅是一条记录。

编写安全的 `SECURITY DEFINER` 函数

因为 `SECURITY DEFINER` 函数是以创建它的用户的权限执行的， 所以一定确保这样的函数不被滥用。为了安全， 应该设置 search_path 排除可以被任何不信任用户更改的模式。 这能避免恶意用户用生成的对象来替代函数中用到的对象的攻击方法。 特别应该注意的是临时表模式，该模式默认在搜索路径中排在第一个，并且对任何用户可写。 为安全考虑可以强制临时表模式最后被搜索。为了达到这个目的，可以将 `pg_temp` 放到 `search_path` 的最后。下面是安全定义函数的一个例子：

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- 设置安全的 search_path: 信任的模式, 然后是 'pg_temp' 模式
    SET search_path = admin, pg_temp;
```

因为 PostgreSQL 8.3 版本没有 `SET` 选项，所以老函数可以包含比较复杂的逻辑去存储，设置和恢复 `search_path` 参数，而使用 `SET` 选项则简单的多。

另一个需要注意的是：默认新创建的函数的执行权限被授予了 `PUBLIC`，（更多信息见 GRANT）。 更常见的是你希望限制安全函数只给某些用户使用。为了实现这个目的，你必须收回默认的给 `PUBLIC` 的权限，并且单独给选定用户分配权限。为消除新函数能被所有人执行的时间窗口，可以在一个事务中创建和设置函数权限，例如：

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

兼容性

PostgreSQL 里的版本和 SQL:1999 里的 `CREATE FUNCTION` 命令类似但是不完全兼容。 属性和可以使用的语言都是不可移植的。

为了和一些其它的数据库系统兼容，`_argmode_` 可以在 `_argname_` 之前或者之后写， 但是只有第一种写法是与标准兼容的。

对于默认参数，SQL标准仅仅规定了使用 `DEFAULT` 关键字的语法。 在T-SQL 和 Firebird 中，也使用 `=` 的语法。

参见

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE, createlang

CREATE GROUP

Name

CREATE GROUP -- 定义一个新数据库角色

Synopsis

```
CREATE GROUP _name_ [ [ WITH ] _option_ [ ... ] ]

这里的`_option_`可以是:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD '_password_'
    | VALID UNTIL '_timestamp_'
    | IN ROLE _role_name_ [, ...]
    | IN GROUP _role_name_ [, ...]
    | ROLE _role_name_ [, ...]
    | ADMIN _role_name_ [, ...]
    | USER _role_name_ [, ...]
    | SYSID _uid_
```

描述

`CREATE GROUP` 现在是CREATE ROLE的别名。

兼容性

SQL标准里没有 `CREATE GROUP` 语句。

又见

CREATE ROLE

CREATE INDEX

Name

CREATE INDEX -- 创建一个索引

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ _name_ ] ON _table_name_ [ USING _method_ ]
    ( { _column_name_ | ( _expression_ ) } [ COLLATE _collation_ ] [ _opclass_ ] [ ASC |
    [ WITH ( _storage_parameter_ = _value_ [, ... ] ) ]
    [ TABLESPACE _tablespace_name_ ]
    [ WHERE _predicate_ ]
```

描述

`CREATE INDEX` 在一个指定表或者物化视图的指定列上创建一个索引,索引主要用来提高数据库的效率(尽管不合理的使用将导致较慢的效率)

索引的键字是用列名来声明的,或者在括号里面写一个表达式 .如果索引支持多列索引,我们也可以指定多个字段.

一个索引域可以是一个使用表的一个或多个字段进行计算的表达式。 这个特性可以快速访问一些基础数据的变形.例如,一个在 `upper(col)` 上计算的索引允许 `WHERE upper(col) = 'JIM'` 字句使用这个索引.

PostgreSQL提供的索引类型包括B-tree, hash, GiST, SP-GiST, 和 GIN.用户也可以自己定义索引类型,但这是相当复杂的.

当使用 `WHERE` 字句,将会创建一个*partial index*.一个部分索引是仅包括表的一部分记录的索引,通常是比表中其他部分更有用的记录. 例如,如果你有一个包含了记账和不记账的订单,不记帐的订单只占了表的一小部分,并且是经常使用的部分.你可以在这部分上建立一个索引来提高效率.另一个应用是使用 `WHERE` 和 `UNIQUE` 来强制保证表的一个子集的唯一性.更多的信息见:Section 11.8

`WHERE` 字句的表达式可以使用底层表的列,并且可以是用它的所有列,而不仅仅建立索引的列. 目前子查询和聚合表达式时禁止出现在 `WHERE` 字句的,这个限制同样适用于索引.

在索引定义中使用的所有函数和操作符必须是"immutable",即他们的结果依赖与他们的参数,而不受任何外部的数据的影响(例如另一个表的内容,或当前时间).这个限制保证了索引的定义完整性.在一个索引或 `WHERE` 子句中使用用户自定义的函数时,确保在定义这些函数时标记他们是不可变的.

Parameters

`UNIQUE`

当索引被创建时使系统检查表的重复值(如果已经有数据),并且当有数据插入时检查唯一性.当插入数据或更新数据导致重复记录时将会产生一个错误.

`CONCURRENTLY`

当使用这个选项时, PostgreSQL在生成索引时将不会在表上加任何锁阻止并发的插入,更新,删除.而标准的建立一个索引将产生一个阻止写(不包括读)的锁直到索引建立完毕.当使用这个选项时有一些问题需要注意,详见:并发建立索引.

`_name_`

被创建索引的名字.这里不需要包括模式的名字,索引总是在同一个模式中作为其父表创建的.如果忽略这个选项,PostgreSQL基于父表的名字和建立索引的列选择一个合适的名字.

`_table_name_`

要建立索引的表名(可能有模式修饰)

`_method_`

建立索引使用的方法名字. 可选的方法有 `btree` , `hash` , `gist` , `spgist` and `gin` .默认方法是 `btree` .

`_column_name_`

表中 要建立索引的列名.

`_expression_`

基于表中一列或多列的表达式.在上面的语法中,表达式必须在园括号中.如果表达式有函数调用的格式则圆括号可以省略.

`_collation_`

索引使用排序方式的名字,默认索引使用创建索引列中声明的排序方式,创建索引的表达式结果的排序方式.对于使用指定排序方式的表达式查询,索引使用指定的排序是高效的.

`_opclass_`

操作符类的名字,详细内容见下面.

`ASC`

指定升序排序(默认).

`DESC`

指定降序排序.

`NULLS FIRST`

指定null排在非null值前面,在 `DESC` 中时默认的.

`NULLS LAST`

指定null排在非null值后面, 默认在 `DESC` 没有指定.

`_storage_parameter_`

index-method-specific 存储参数的名字.详细见:索引存储参数.

`_tablespace_name_`

在哪个表空间建立索引,如果不指定,将使用default_tablespace ,或者临时表的索引将使用temp_tablespaces .

`_predicate_`

部分索引的约束表达式.

索引存储参数

可选的 `WITH` 子句指定了索引的*storage parameters*.每个索引方式有它自己的存储参数.B-tree, hash, GiST and SP-GiST索引都可以带这个参数.

`FILLFACTOR`

填充因子是一个索引实际数据的百分比,它决定索引方法占用页的空间的比率.对于B-trees,在索引创建时叶子页填充这个百分比的数据.其余的用来扩展索引(添加一个新的最大的键值).如果接下来页占用100%,它将会分页,这将导致索引效率的逐渐下降.B-tree使用一个90做为默认填充比,但是可以选择10到100的任何值.如果是一个静态表,填充比使用100将会最大的减小索引所占的物理空间.但是对于有大量更新的表一个较小的填充比将会尽可能的减少分页.其他的索引方法使用填充比是不同的但是大致方法是相同; 方法之间的默认填充比不是一致的.

GiST索引额外接受这个参数:

`BUFFERING`

在使用Section 55.3.1建立索引时决定是否缓存建立的方法.使用 `OFF` 关闭这个功能, `ON` 打开这个功能. 使用 `AUTO` 它初始化时是关闭的,但是当索引的达到effective_cache_size将会打开. 默认使用 `AUTO` .

GIN索引接受一个不同的参数:

`FASTUPDATE`

这个设置用来控制在 Section 57.3.1中描述的快速更新技术.它是一个布尔类型参数: `ON` 使能快速更新, `OFF` 关闭这个功能.(在Section 18.1中描述了允许可选的拼写 `ON` and `OFF` .)默认是 `ON` .

> **Note:** 通过 `ALTER INDEX` 关闭 `FASTUPDATE` 防止将来数据插入到待建立的索引记录中,但是它自己不写之前的记录到磁盘.你可以使用 `VACUUM` 表来确保待建立索引空.

并发建立索引

建立一个索引将影响正常的数据库操作.一般的 PostgreSQL锁住建立索引的表防止写，然后通过扫描表来建立整个索引.其他的事物可以读表,但是插入,更新,删除操作将被锁住直到索引建立完成.如果这是一个线上的生成库将会有较严重的影响.非常大的表将使用数个小时来建立索引,即使一个较小的表,对于生产库也会在一个不可接受的时间内锁住该表的写操作.

PostgreSQL支持建立索引时不锁写操作.这个方法通过 `CREATE INDEX` 时指定 `CONCURRENTLY` 选项,当使用这个选项时, PostgreSQL必须扫描表2次,它必须等待所有的将要使用该索引的事物结束.所以这个方法比标准的建立索引需要更多的工作并且花费更多的时间.然而,在建立索引的时候它允许正常的操作所以对于生产环境它是非常有用的.当然在建立索引时额外的CPU和I/o开销可能会降低其他操作的效率.

在并发创建索引中，索引在一个事物中进入系统表，然后两表扫描发生在另外的事物中．在第二遍扫描时任何活跃的事物（在该表上）将会阻塞并发索引的建立直到事物完成．甚至在第二遍表扫描时事物仅仅涉及到该表．并发建立索引使用Section 47.59中的方法等待每一个发生的事物完成．

当扫描表时发生了问题，例如一个唯一索引违反了唯一性， `CREATE INDEX` 将会失败并且留下一个 "invalid"索引．在查询时这个索引将会被忽略因为他是不完整的；然而它仍然会增加更新的开销．psql `\d` 命令将列出带 `INVALID` 的索引．

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col) INVALID
```

这种情况推荐的恢复方法是删除掉索引并且再次执行 `CREATE INDEX CONCURRENTLY` ．（另一种重建索引的方法是使用 `REINDEX` .然而，因为 `REINDEX` 不支持并发建立索引，所以这种方式可用性不高．）

另一个缺点是当并发建立唯一索引第二遍扫描表后唯一性将会约束其他事物．这意味着在索引建立完成前在其他查询中违反了该索引将会报违反约束错误，即使最后索引建立失败．如果在第二遍扫描发生了错误，在后续的操作中"invalid"索引也会（在相关列上）保持该唯一性．

支持建立并发索引和部分索引. 在这些语句中出现错误将会产生和前面违反唯一性约束相似的错误.

标准建立索引允许并行的执行其他标准的建立索引语句, 但是在一个表上一次仅可以有一个并发建立索引语句. 在这两种情况下, 建立索引的同时不允许有其他模式类型的修改. 另一个不同点是 `CREATE INDEX` 可以在一个阻塞的事物里执行, 而 `CREATE INDEX CONCURRENTLY` 不可以.

Notes

在Chapter 11中描述了什么时候会使用索引, 什么时候不使用索引, 以及在什么情况下索引是有用的.

> **Caution**
>
> 哈希索引不记录到WAL日志中, 所以当系统崩溃有未写入磁盘的数据时哈希索引需要使用 `REINDEX` 重新建立. 哈希索引的变化不能在恢复一个基础备份后通过流复制或者文件复制来重写. 所以接下来在查询中使用他们将会给出错误的结果. 因为这些原因使用哈希索引是有阻碍的.

现在只有B-tree, GiST 和 GIN支持多列索引. 默认支持最多32列. (当编译PostgreSQL时可以改变这个限制). 现在只有B-tree支持唯一索引.

对于索引的每一列可以指定一个*operator class*. *operator class*标识了索引那一列的使用的操作符. 例如一个B-tree索引在一个四字节整数上可以使用 `int4_ops` ; 这个操作符类包括四字节整数的比较函数. 实际上对于列上的数据类型默认的操作符类时足够用的. 操作符类主要用于一些有多种排序的数据. 例如, 我们想按照绝对值或者实数部分排序一个复数. 我们能通过定义两个操作符类然后当建立索引时选择合适的类. 详细的操作符类信息请参考 : Section 11.9和Section 35.14.

索引 (现在只有B-tree) 中支持有序扫描的字句中可以制定 `ASC` , `DESC` , `NULLS FIRST` , 和 `NULLS LAST` 修改索引的排序方式. 因为一个有序的索引能被从前向后或者从后向前扫描. 建立一个单列的 `DESC` 索引是没有意义的. 因为在一个标准的索引是已经排序的. 在创建多列索引时使用这些选项值来匹配一些复合查询的排序请求, 例如 `SELECT ... ORDER BY x ASC, y DESC` .如果你在查询中需要支持 "nulls sort low"特性 `NULLS` 选项是很有用的, 而不是默认的"nulls sort high",

对于大多数索引, 创建索引的速度依赖于maintenance_work_mem的设置. 只要你没有使他超过可得到内存而进入swap分区, 较大的值将会减少索引创建的时间. 对于创建哈希索引的时间与effective_cache_size值相关,PostgreSQL将使用两种不同的哈希索引创建方法, 主要取决于索引的大小比 `effective_cache_size` 大还是小. 为了得到较好的结果, 确保这个参数的设置考虑到可用内存, 并且注意 `maintenance_work_mem` 和 `effective_cache_size` 的和小于其他程序所需要的内存空间.

使用 DROP INDEX删除一个索引.

PostgreSQL之前的版本有一个R-tree索引方法. 这个方法已经被删除因为它对于GiST索引没有明显的优势. 如果指定 `USING rtree` , `CREATE INDEX` 会将他翻译为 `USING gist` , 去转换老的数据库版本到GiST.

Examples

在表 `films` 的 `title` 列上建立B-tree索引：

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

允许大小写无关查询在 `lower(title)` 表达式上建立索引：

```
CREATE INDEX ON films ((lower(title)));
```

(在这个例子中我们可以选在忽略索引名字, 系统将给出一个名字, 典型的是 `films_lower_idx` .)

创建一个使用非默认排序方式的索引：

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

创建一个null值的非默认排序的索引：

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

创建一个非默认填充因子的索引：

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

创建一个将快速更新关闭的GIN索引：

```
CREATE INDEX gin_idx ON documents_table USING gin (locations) WITH (fastupdate = off);
```

创建一个在 `films` 的 `code` 列的索引, 并且使索引建立在 `indexspace` 表空间中：

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

在一个点属性上创建GiST索引, 让我们在转换函数的结果上高效的使用box操作符：

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

创建一个在表上不加写锁的索引：

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` 是PostgreSQL语言的扩展。在SQL标准中没有索引的规定。

See Also

ALTER INDEX, DROP INDEX

CREATE LANGUAGE

Name

CREATE LANGUAGE -- define a new procedural language

Synopsis

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE _name_
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE _name_
    HANDLER _call_handler_ [ INLINE _inline_handler_ ] [ VALIDATOR _valfunction_ ]
```

Description

`CREATE LANGUAGE` registers a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language.

> **Note:** As of PostgreSQL 9.1, most procedural languages have been made into "extensions", and should therefore be installed with CREATE EXTENSION not `CREATE LANGUAGE`. Direct use of `CREATE LANGUAGE` should now be confined to extension installation scripts. If you have a "bare" language in your database, perhaps as a result of an upgrade, you can convert it to an extension using `CREATE EXTENSION` `_langname_` FROM unpackaged.

`CREATE LANGUAGE` effectively associates the language name with handler function(s) that are responsible for executing functions written in the language. Refer to Chapter 51 for more information about language handlers.

There are two forms of the `CREATE LANGUAGE` command. In the first form, the user supplies just the name of the desired language, and the PostgreSQL server consults the `pg_pltemplate` system catalog to determine the correct parameters. In the second form, the user supplies the language parameters along with the language name. The second form can be used to create a language that is not defined in `pg_pltemplate`, but this approach is considered obsolescent.

When the server finds an entry in the `pg_pltemplate` catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Ordinarily, the user must have the PostgreSQL superuser privilege to register a new language. However, the owner of a database can register a new language within that database if the language is listed in the `pg_pltemplate` catalog and is marked as allowed to be created by database owners (`tmpldbacreate` is true). The default is that trusted languages can be created by database owners, but this can be adjusted by superusers by modifying the contents of `pg_pltemplate` . The creator of a language becomes its owner and can later drop it, rename it, or assign it to a new owner.

`CREATE OR REPLACE LANGUAGE` will either create a new language, or replace an existing definition. If the language already exists, its parameters are updated according to the values specified or taken from `pg_pltemplate` , but the language's ownership and permissions settings do not change, and any existing functions written in the language are assumed to still be valid. In addition to the normal privilege requirements for creating a language, the user must be superuser or owner of the existing language. The `REPLACE` case is mainly meant to be used to ensure that the language exists. If the language has a `pg_pltemplate` entry then `REPLACE` will not actually change anything about an existing definition, except in the unusual case where the `pg_pltemplate` entry has been modified since the language was created.

Parameters

`TRUSTED`

`TRUSTED` specifies that the language does not grant access to data that the user would not otherwise have. If this key word is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

`PROCEDURAL`

This is a noise word.

`_name_`

The name of the new procedural language. The name must be unique among the languages in the database.

For backward compatibility, the name can be enclosed by single quotes.

`HANDLER` `_call_handler_`

`_call_handler_` is the name of a previously registered function that will be called to execute the procedural language's functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the `language_handler` type, a placeholder type that is simply used to identify the function as a call handler.

`INLINE` `_inline_handler_`

`_inline_handler_` is the name of a previously registered function that will be called to execute an anonymous code block (DO command) in this language. If no `_inline_handler_` function is specified, the language does not support anonymous code blocks. The handler function must take one argument of type `internal`, which will be the `DO` command's internal representation, and it will typically return `void`. The return value of the handler is ignored.

`VALIDATOR` `_valfunction_`

`_valfunction_` is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is specified, then a new function will not be checked when it is created. The validator function must take one argument of type `oid`, which will be the OID of the to-be-created function, and will typically return `void`.

A validator function would typically inspect the function body for syntactical correctness, but it can also look at other properties of the function, for example if the language cannot handle certain argument types. To signal an error, the validator function should use the `ereport()` function. The return value of the function is ignored.

The `TRUSTED` option and the support function name(s) are ignored if the server has an entry for the specified language name in `pg_pltemplate`.

Notes

The createlang program is a simple wrapper around the `CREATE LANGUAGE` command. It eases installation of procedural languages from the shell command line.

Use DROP LANGUAGE, or better yet the droplang program, to drop procedural languages.

The system catalog `pg_language` (see Section 47.28) records information about the currently installed languages. Also, `createlang` has an option to list the installed languages.

To create functions in a procedural language, a user must have the `USAGE` privilege for the language. By default, `USAGE` is granted to `PUBLIC` (i.e., everyone) for trusted languages. This can be revoked if desired.

Procedural languages are local to individual databases. However, a language can be installed into the `template1` database, which will cause it to be available automatically in all subsequently-created databases.

The call handler function, the inline handler function (if any), and the validator function (if any) must already exist if the server does not have an entry for the language in `pg_pltemplate`. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database. (This might result in `CREATE LANGUAGE` failing, if the shared library that implements the language is not available in the installation.)

In PostgreSQL versions before 7.3, it was necessary to declare handler functions as returning the placeholder type `opaque`, rather than `language_handler`. To support loading of old dump files, `CREATE LANGUAGE` will accept a function declared as returning `opaque`, but it will issue a notice and change the function's declared return type to `language_handler`.

Examples

The preferred way of creating any of the standard procedural languages is just:

```
CREATE LANGUAGE plperl;
```

For a language not known in the `pg_pltemplate` catalog, a sequence such as this is needed:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

`CREATE LANGUAGE` is a PostgreSQL extension.

See Also

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE, createlang, droplang

CREATE MATERIALIZED VIEW

Name

CREATE MATERIALIZED VIEW -- 定义一个物化视图

Synopsis

```
CREATE MATERIALIZED VIEW _table_name_
    [ (_column_name_ [, ...] ) ]
    [ WITH ( _storage_parameter_ [= _value_] [, ... ] ) ]
    [ TABLESPACE _tablespace_name_ ]
    AS _query_
    [ WITH [ NO ] DATA ]
```

描述

`CREATE MATERIALIZED VIEW` 定义一个查询的物化视图。 在命令发出时，该查询被执行并且用于填充视图（除非使用了 `WITH NO DATA` ）， 并且可以稍后使用 `REFRESH MATERIALIZED VIEW` 来刷新。

`CREATE MATERIALIZED VIEW` 类似于 `CREATE TABLE AS`， 除了它也使查询记住初始化视图，所以稍后需要时也可以被刷新。 物化视图和表一样有许多属性，但是不支持临时物化视图或自动生成OID。

参数

`_table_name_`

要创建的物化视图的名字（可以有模式修饰）。

`_column_name_`

新的物化视图中的字段名。如果没有提供字段名，那么使用查询的输出字段名。

`WITH (` `_storage_parameter_` [= `_value_`] [, ...])

这个子句为新的物化视图指定可选的存储参数；参阅 存储参数 获取更多信息。所有 `CREATE TABLE` 支持的参数， `CREATE MATERIALIZED VIEW` 也都支持，除了 `OIDS` 。 参阅CREATE TABLE获取更多信息。

`TABLESPACE` `_tablespace_name_`

`_tablespace_name_` 是新的物化视图在其中创建的表空间的名字。如果没有指定， 则查询 default_tablespace。

`_query_`

SELECT、TABLE或 VALUES命令。这个查询将在受到安全限制的操作内运行； 特别的，对函数的调用本身创建临时表将会失败。

`WITH [ NO ] DATA`

这个子句说明物化视图是否在创建时填充。如果不，物化视图将被标记为不可扫描， 并且不能被查询， 直到使用了 `REFRESH MATERIALIZED VIEW` 。

兼容性

`CREATE MATERIALIZED VIEW` 是一个PostgreSQL扩展。

又见

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

Name

CREATE OPERATOR -- 定义一个新操作符

Synopsis

```
CREATE OPERATOR _name_ (
    PROCEDURE = _function_name_
    [, LEFTARG = _left_type_ ] [, RIGHTARG = _right_type_ ]
    [, COMMUTATOR = _com_op_ ] [, NEGATOR = _neg_op_ ]
    [, RESTRICT = _res_proc_ ] [, JOIN = _join_proc_ ]
    [, HASHES ] [, MERGES ]
)
```

描述

`CREATE OPERATOR` 定义一个新的 `_name_` 操作符。 定义该操作符的用户将成为其所有者。如果给出了一个模式名， 那么该操作符将在指定的模式中创建。否则它会在当前模式中创建。

操作符 name 是一个最多 `NAMEDATALEN` -1长的(缺省为 63 个) 下列字符组成的字符串：

-
    -
        - / < > = ~ ! @ # % ^ & | ` ?

你选择名字的时候有几个限制：

- `--` 和 `/*` 不能在操作符名的任何地方出现， 因为它们会被认为是一个注释的开始。
- 一个多字符的操作符不能以 `+` 或 `-` 结尾， 除非该名字还包含至少下面字符之一：

  ~ ! @ # % ^ & | ` ?

  例如， `@-` 是一个允许的操作符名，但 `*-` 不是。 这个限制允许PostgreSQL 分析 SQL 兼容的查询而不要求在符号之间有空白。

- => 作为一个操作符名的使用已经废弃了。可能在未来的版本中完全禁用。

操作符 `!=` 在输入时映射成 <> ， 因此这两个名称总是等价的。

至少需要定义一个 `LEFTARG` 和 `RIGHTARG` 。对于双目操作符来说， 两者都需要定义。对右目操作符来说，只需要定义 `LEFTARG` ， 而对于左目操作符来说，只需要定义 `RIGHTARG` 。

同样， `_function_name_` 过程必须已经用 `CREATE FUNCTION` 定义过， 而且必须定义为接受正确数量的指定类型参数(一个或是两个)。

其它子句声明可选的操作符优化子句。他们的含义在Section 35.13里定义。

要想能够创建一个操作符，你必须在参数类型和返回类型上有 `USAGE` 权限， 还要在底层函数上有 `EXECUTE` 权限。如果指定了交换或者负操作符， 你必须拥有这些操作符。

参数

`_name_`

要定义的操作符。可用的字符见上文。其名字可以用模式修饰， 比如 `CREATE OPERATOR myschema.+ (...)` 。如果没有模式， 则在当前模式中创建操作符。同一个模式中的两个操作符可以有一样的名字， 只要他们操作不同的数据类型。这叫做重载。

`_function_name_`

用于实现该操作符的函数。

`_left_type_`

操作符左边的参数数据类型，如果存在的话。如果是左目操作符，这个参数可以省略。

`_right_type_`

操作符右边的参数数据类型，如果存在的话。如果是右目操作符，这个参数可以省略。

`_com_op_`

该操作符对应的交换操作符。

`_neg_op_`

该操作符对应的负操作符。

`_res_proc_`

此操作符约束选择性评估函数。

`_join_proc_`

此操作符连接选择性评估函数。

`HASHES`

表明此操作符支持 Hash 连接。

`MERGES`

表明此操作符可以支持一个融合连接。

使用 `OPERATOR()` 语法在 `_com_op_` 或者其它可选参数里给出一个模式修饰的操作符名，比如：

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

注意

参阅Section 35.12中的操作符章节获取更多信息。

在 `CREATE OPERATOR` 中指定操作符的词法优先级是不可能的， 因为分析器的优先级行为是硬链接的。参阅Section 4.1.6 获取优先级的详细信息。

废弃的选项 `SORT1` 、 `SORT2` 、 `LTCMP` 和 `GTCMP` 以前用于指定与可合并连接的操作符相关的排序操作符的名字。 现在不再需要了，因为相关操作符的信息通过查询B-tree操作符类来找到。 如果给出了这些中的一个选项，那么会忽略该选项，除非暗中设置 `MERGES` 为真。

使用DROP OPERATOR从数据库中删除用户定义操作符。 使用ALTER OPERATOR修改一个数据库里的操作符。

例子

下面命令定义一个新操作符：面积相等，用于 `box` 数据类型。

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

兼容性

`CREATE OPERATOR` 是PostgreSQL扩展。 SQL 标准中没有该语句。

又见

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS -- 定义一个新操作符类

Synopsis

```
CREATE OPERATOR CLASS _name_ [ DEFAULT ] FOR TYPE _data_type_
  USING _index_method_ [ FAMILY _family_name_ ] AS
  {  OPERATOR _strategy_number_ _operator_name_ [ ( _op_type_, _op_type_ ) ] [ FOR SEARCH
   | FUNCTION _support_number_ [ ( _op_type_ [ , _op_type_ ] ) ] _function_name_ ( _argum
   | STORAGE _storage_type_
  } [, ... ]
```

描述

`CREATE OPERATOR CLASS` 定义一个新的操作符类。 一个操作符类定义一种特定的数据类型如何与一种索引一起使用。 操作符类声明特定的操作符可以为这种数据类型以及这种索引方法填充特定的角色或者 "策略"。操作符类还声明索引方法在为一个索引字段选定该操作符类的时候要 使用的支持过程。所有操作符类使用的函数和操作符都必须在创建操作符类之前定义。

如果给出了模式名字，那么操作符类就在指定的模式中创建。 否则就在当前模式中创建。在同一个模式中的两个操作符类可以有同样的名字， 但它们必须用于不同的索引方法。

定义操作符类的用户将成为其所有者。目前，创造者必须是超级用户。 做这样的限制是因为一个有问题的操作符类定义会让服务器困惑，甚至崩溃。

`CREATE OPERATOR CLASS` 既不检查这个类定义是否包含所有索引方法需要的操作符以及函数，也不检查这些操作符和函数是否形成一个自包含的集合。 定义一个合法的操作符类是用户的责任。

相关的操作符类可以集合成操作符族。 添加一个新的操作符类到一个已经存在的操作符族，在 `CREATE OPERATOR CLASS` 中指定 `FAMILY` 选项。 没有这个选项，新建的类会放置到与它同名的族中（如果不存在则创建它）。

参考Section 35.14获取更多信息。

参数

`_name_`

将要创建的操作符类的名字(可以用模式修饰)。

`DEFAULT`

表示该操作符类将成为它的数据类型的缺省操作符类。 对于某个数据类型和访问方式而言，最多有一个操作符类是缺省的。

`_data_type_`

这个操作符类处理的字段的数据类型。

`_index_method_`

这个操作符类处理的索引方法的名字。

`_family_name_`

这个操作符类添加到的现有操作符族的名字。如果没有指定， 则使用与该操作符类相同名字的操作符族（如果不存在则创建它）。

`_strategy_number_`

一个操作符和这个操作符类关联的索引方法的策略数。

`_operator_name_`

一个和该操作符类关联的操作符的名字(可以用模式修饰)。

`_op_type_`

在 `OPERATOR` 子句中，该操作符的操作数的数据类型， 或者是 `NONE` 表示左目或者右目操作符。 通常情况下可以省略操作数的数据类型，因为这个时候它们和操作符类的数据类型相同。

在 `FUNCTION` 子句中，如果函数的操作数数据类型和函数的输入数据类型 （对于B-tree比较函数和哈希函数）或类的数据类型 （对于B-tree排序支持函数和所有在GiST、SP-GiST和GIN操作符类中的函数）不同， 那么就在该子句中写上这个函数要支持的操作数类型。这些缺省是正确的， 因此 `_op_type_` 不需要在 `FUNCTION` 子句中指定， 除了B-tree排序支持函数支持交叉数据类型比较的情况。

`_sort_family_name_`

描述与排序操作符相关的排序顺序的现有 `btree` 操作符族的名字（可以有模式修饰）。

如果既没有指定 `FOR SEARCH` ， 也没有指定 `FOR ORDER BY` ， 那么缺省是 `FOR SEARCH` 。

`_support_number_`

索引方法对一个与操作符类关联的函数的支持过程数

`_function_name_`

一个函数的名字(可以有模式修饰)，这个函数是索引方法对此操作符类的支持过程。

`_argument_type_`

函数参数的数据类型。

`_storage_type_`

实际存储在索引里的数据类型。通常它和字段数据类型相同， 但是一些索引方法(目前是GiST和GIN)允许它是不同的。 除非索引方法允许使用一种不同的类型，否则必须省略 `STORAGE` 子句。

`OPERATOR` 、 `FUNCTION` 和 `STORAGE` 子句可以按照任意顺序出现。

注意

因为索引机制不在使用函数前检查其访问机制， 在操作符类中包含操作符或者函数等价于授权给所有人执行权限。 这对于那些用于操作符类的函数通常不会导致什么问题。

操作符不应该用 SQL 函数定义。一个 SQL 函数很可能是内联到调用它的查询里面， 这样将阻止优化器识别这个查询是否可以使用索引。

PostgreSQL 8.4之前， `OPERATOR` 子句可以包括 `RECHECK` 选项。现在不再支持了， 因为一个索引操作符是否是"有损耗的"，现在是在运行时动态确定。 这允许高效的处理操作符有或没有损耗的情况。

例子

下面的例子命令为数据类型 `_int4` (`int4` 数组) 定义了一个 GiST 索引操作符类。参阅intarray模块获取完整的例子。

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

兼容性

`CREATE OPERATOR CLASS` 是一个PostgreSQL 扩展。在SQL标准中没有这个语句。

又见

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

Name

CREATE OPERATOR FAMILY -- 定义一个新操作符族

Synopsis

```
CREATE OPERATOR FAMILY _name_ USING _index_method_
```

描述

`CREATE OPERATOR FAMILY` 创建一个新的操作符族。 一个操作符族定义一个相关的操作符类的集合， 或许还有与这些操作符类兼容但对单独索引的运行不重要的一些额外的操作符和支持函数。 （对索引来说重要的操作符和函数应该分组到相关的操作符类中， 而不是"散漫"在操作符族中。典型的，单数据类型操作符绑定到操作符类中， 而交叉数据类型操作符可以散漫在一个包含两种数据类型的操作符类的操作符族中。）

新的操作符族初始为空。应该随后发出 `CREATE OPERATOR CLASS` 命令来添加所包含的操作符类，和可选的 `ALTER OPERATOR FAMILY` 命令来添加"散漫的"操作符和它们对应的支持函数。

如果给出了模式名，则操作符族在指定的模式中创建。否则在当前模式中创建。 同一个模式中的两个操作符族可以有相同的名字，只要他们处理的索引方法不同就可以。

定义操作符族的用户成为其所有者。目前，创建操作符族的用户必须是超级用户。 （做这个限制是因为错误的操作符族的定义会导致服务器混乱，甚至崩溃。）

参阅Section 35.14获取更多信息。

参数

`_name_`

要创建的操作符族的名字。该名字可以有模式修饰。

`_index_method_`

这个操作符族处理的索引方法的名字。

兼容性

`CREATE OPERATOR FAMILY` 是一个PostgreSQL 扩展。在 SQL 标准中没有这个语句。

又见

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE ROLE

Name

CREATE ROLE -- 定义一个新的数据库角色

Synopsis

```
CREATE ROLE _name_ [ [ WITH ] _option_ [ ... ] ]

where `_option_` can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT _connlimit_
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD '_password_'
    | VALID UNTIL '_timestamp_'
    | IN ROLE _role_name_ [, ...]
    | IN GROUP _role_name_ [, ...]
    | ROLE _role_name_ [, ...]
    | ADMIN _role_name_ [, ...]
    | USER _role_name_ [, ...]
    | SYSID _uid_
```

Description

`CREATE ROLE` 命令用于为PostgreSQL数据库集群增加新的角色。 角色是拥有数据库对象和数据库权限的实体。一个角色可以依据其被使用的情况，被看做一个用户"user"或者一个组"group"。 参考Chapter 20和Chapter 19这两个文档，去获取有关用户和认证管理的相关信息。 你必须要有 `CREATEROLE` 权限，或者你是一个超级用户，你才能使用 `CREATE ROLE` 命令。

注意，角色是定义在数据库集群级别的，所以在集群中的所有数据库中都是有效的。

参数

`_name_`

新角色的名称。

`SUPERUSER``NOSUPERUSER`

这两个条件决定一个新的角色是否为一个超级用户（"superuser"），超级用户可以超越数据库内的所有访问限制。 超级用户状态是危险的，应该在真正需要的情况下才被使用。你自己必须是一个超级用户，才能创建一个新的超级用户。如果没有指定 这个条件，缺省是 `NOSUPERUSER` 。

`CREATEDB` `NOCREATEDB`

这两个条件定义用户创建数据库的权限。如果被指定为 `CREATEDB`，则该用户被赋予创建数据库的权限。 被指定为 `NOCREATEDB` 将没有创建数据库的权限。如果没有指定这个条件，缺省是 `NOCREATEDB` 。

`CREATEROLE` `NOCREATEROLE`

这两个条件决定一个角色是否被允许创建新角色（即，执行 `CREATE ROLE` 命令的权限）。 一个角色如果被赋予 `CREATEROLE` 权限，则同时也具有了修改或删除其他角色的权限。如果没有指定这个条件， 则缺省是 `NOCREATEROLE` 。

`CREATEUSER` `NOCREATEUSER`

这两个条件是过时的，不过现在仍被数据库接受，他们和条件 `SUPERUSER` `NOSUPERUSER` 是等效的。 注意，他们和 `CREATEROLE` `NOCREATEROLE` 并无关系。

`INHERIT` `NOINHERIT`

These clauses determine whether a role "inherits" the privileges of roles it is a member of. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. Without `INHERIT` , membership in another role only grants the ability to `SET ROLE` to that other role; the privileges of the other role are only available after having done so. If not specified, `INHERIT` is the default.

`LOGIN` `NOLOGIN`

These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, `NOLOGIN` is the default, except when `CREATE ROLE` is invoked through its alternative spelling CREATE USER. 这两个参数决定一个角色是否有登进数据库的权限；一个拥有 `LOGIN` 属性的角色（role），可以被视为一个用户（user）。

`REPLICATION` `NOREPLICATION`

These clauses determine whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the `REPLICATION` attribute is a very highly privileged role, and should only be used on roles actually used for replication. If not

specified, `NOREPLICATION` is the default.

`CONNECTION LIMIT` `_connlimit_`

If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.

`PASSWORD` `_password_`

Sets the role's password. (A password is only of use for roles having the `LOGIN` attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

`ENCRYPTED` `UNENCRYPTED`

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter password_encryption.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

Note that older clients might lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

`VALID UNTIL` ' `_timestamp_` '

The `VALID UNTIL` clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

`IN ROLE` `_role_name_`

The `IN ROLE` clause lists one or more existing roles to which the new role will be immediately added as a new member. (Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.)

`IN GROUP` `_role_name_`

`IN GROUP` is an obsolete spelling of `IN ROLE`.

`ROLE` `_role_name_`

The `ROLE` clause lists one or more existing roles which are automatically added as members of the new role. (This in effect makes the new role a "group".)

`ADMIN` `_role_name_`

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

`USER` `_role_name_`

The `USER` clause is an obsolete spelling of the `ROLE` clause.

`SYSID` `_uid_`

The `SYSID` clause is ignored, but is accepted for backwards compatibility.

Notes

Use ALTER ROLE to change the attributes of a role, and DROP ROLE to remove a role. All the attributes specified by `CREATE ROLE` can be modified by later `ALTER ROLE` commands.

The preferred way to add and remove members of roles that are being used as groups is to use GRANT and REVOKE.

The `VALID UNTIL` clause defines an expiration time for a password only, not for the role *per se*. In particular, the expiration time is not enforced when logging in using a non-password-based authentication method.

The `INHERIT` attribute governs inheritance of grantable privileges (that is, access privileges for database objects and role memberships). It does not apply to the special role attributes set by `CREATE ROLE` and `ALTER ROLE`. For example, being a member of a role with `CREATEDB` privilege does not immediately grant the ability to create databases, even if `INHERIT` is set; it would be necessary to become that role via SET ROLE before creating a database.

The `INHERIT` attribute is the default for reasons of backwards compatibility: in prior releases of PostgreSQL, users always had access to all privileges of groups they were members of. However, `NOINHERIT` provides a closer match to the semantics specified in the SQL standard.

Be careful with the `CREATEROLE` privilege. There is no concept of inheritance for the privileges of a `CREATEROLE`-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if the role "user" has the `CREATEROLE` privilege but not the `CREATEDB` privilege, nonetheless it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

PostgreSQL includes a program createuser that has the same functionality as `CREATE ROLE` (in fact, it calls this command) but can be run from the command shell.

The `CONNECTION LIMIT` option is only enforced approximately; if two new sessions start at about the same time when just one connection "slot" remains for the role, it is possible that both will fail. Also, the limit is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client's command history or the server log. The command createuser, however, transmits the password encrypted. Also, psql contains a command `\password` that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role with a password:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(`CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN` .)

Create a role with a password that is valid until the end of 2004. After one second has ticked in 2005, the password is no longer valid.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Create a role that can create databases and manage roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibility

The `CREATE ROLE` statement is in the SQL standard, but the standard only requires the syntax

```
CREATE ROLE _name_ [ WITH ADMIN _role_name_ ]
```

Multiple initial administrators, and all the other options of `CREATE ROLE` , are PostgreSQL extensions.

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by each database implementation. In PostgreSQL we have chosen to unify users and roles into a single kind of entity. Roles therefore have many more optional attributes than they do in the standard.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

Name

CREATE RULE -- 定义一个新重写规则

Synopsis

```
CREATE [ OR REPLACE ] RULE _name_ AS ON _event_
    TO _table_name_ [ WHERE _condition_ ]
    DO [ ALSO | INSTEAD ] { NOTHING | _command_ | ( _command_ ; _command_ ... ) }
```

描述

`CREATE RULE` 定义一个适用于特定表或者视图的新规则。 `CREATE OR REPLACE RULE` 要么是创建一个新规则， 要么是替换一个表上的同名规则。

PostgreSQL规则系统允许在更新、插入、 删除时执行一个其它的预定义动作。简单的说，规则就是在指定表上执行指定动作的时候，将导致一些额外的动作被执行。 另外，一个 `INSTEAD` 规则可以用另外一个命令取代特定的命令， 或者令完全不执行该命令。规则还可以用于实现SQL视图。 规则实际上只是一个命令转换机制，或者说命令宏。 这种转换发生在命令开始执行之前。如果你想要针对每个物理行独立发生的操作， 那么可能应该使用触发器而不是规则。有关规则系统的更多信息可以在 Chapter 38找到。

目前， `ON SELECT` 规则必须是无条件的 `INSTEAD` 规则并且必须有一个由单独一条 `SELECT` 查询组成的动作。 因此，一条 `ON SELECT` 规则有效地把表转成视图， 它的可见内容是规则的 `SELECT` 查询返回的记录而不是存储在表（如果有）中的内容。写一条 `CREATE VIEW` 命令比创建一个表然后在上面定义一条 `ON SELECT` 规则的风格要好。

你可以创建一个允许更新的视图的幻觉，方法是在视图上定义 `ON INSERT` 、 `ON UPDATE` 、 `ON DELETE` 规则 (或者满足你需要的任何上述规则的子集)，用合适的对其它表的更新替换在视图上更新的动作。 如果打算支持 `INSERT RETURNING` 之类， 就必须确保在规则的结尾放置恰当的 `RETURNING` 子句。

如果你想在复杂的视图更新上使用条件规则，那么这里就有一个补充： 对你希望在视图上允许的每个动作，你都必须 有一个无条件的 `INSTEAD` 规则。如果规则是有条件的或者它不是 `INSTEAD` ，那么系统仍将拒绝执行更新动作， 因为它认为最终会在视图的虚拟表上执行这个动作。 如果你想处理条件规则上的所有有用的情况，那只需要增加一个无条件的

`DO INSTEAD NOTHING` 规则确保系统明白它决不会被调用来更新虚拟表就可以了。 然后把条件规则做成非 `INSTEAD` ；在这种情况下， 如果它们被触发，那么它们就增加到缺省的 `INSTEAD NOTHING` 动作中。 （不过这种方法目前不支持 `RETURNING` 查询。）

> **Note:** 一个足够简单可以自动更新的视图（参阅CREATE VIEW） 不需要用户创建的使其可更新的规则。不过，你可以创建一个明确的规则， 自动更新转换通常比明确的规则执行的更好。
>
> 另一个可替换的价值考虑是使用 `INSTEAD OF` 触发器 （参阅CREATE TRIGGER）替代规则。

参数

`_name_`

创建的规则名。它必须在同一个表上的所有规则名字中唯一。 同一个表上的同一个事件类型的规则是按照字母顺序运行的。

`_event_`

`SELECT` 、 `INSERT` 、 `UPDATE` 、 `DELETE` 事件之一。

`_table_name_`

规则作用的表或者视图的名字(可以有模式修饰)。

`_condition_`

任意返回 `boolean` 的SQL条件表达式。 条件表达式除了引用 `NEW` 和 `OLD` 之外不能引用任何表， 并且不能有聚集函数。

`INSTEAD`

`INSTEAD` 指示使用该命令代替最初的命令。

`ALSO`

`ALSO` 指示该命令应该在最初的命令执行之后 一起执行。

如果既没有声明 `ALSO` 也没有声明 `INSTEAD` ， 那么 `ALSO` 是缺省。

`_command_`

组成规则动作的命令。有效的命令是 `SELECT` 、 `INSERT` 、 `UPDATE` 、 `DELETE` 、 `NOTIFY` 语句之一。

在 `_condition_` 和 `_command_` 里， 特殊的表名字 `NEW` 和 `OLD` 可以用于指向被引用表里的数值。 `NEW` 在 `ON INSERT` 和 `ON UPDATE` 规则里可以指向被插入或更新的新行。 `OLD` 在 `ON UPDATE` 和 `ON DELETE` 规则里可以指向现存的被更新或删除的行。

注意

为了在表上定义或修改规则，你必须是该表的拥有者。

在视图上用于 `INSERT` 、 `UPDATE` 、 `DELETE` 的规则中可以添加 `RETURNING` 子句基于视图的字段返回。如果规则被 `INSERT RETURNING` 、 `UPDATE RETURNING` 、 `DELETE RETURNING` 命令触发，这些子句将用来计算输出结果。如果规则被不带 `RETURNING` 的命令触发，那么规则的 `RETURNING` 子句将被忽略。 目前仅允许无条件的 `INSTEAD` 规则包含 `RETURNING` 子句， 而且在同一个事件内的所有规则中最多只能有一个 `RETURNING` 子句。 这样就确保只有一个 `RETURNING` 子句可以用于计算结果。 如果在任何有效规则中都不存在 `RETURNING` 子句， 该视图上的 `RETURNING` 查询将被拒绝。

有一件很重要的事情是要避免循环规则。比如，尽管下面两条规则定义都是 PostgreSQL可以接受的，但其中一条的 `SELECT` 命令会导致PostgreSQL 报告一条错误信息，因为该查询循环了太多次：

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

目前，如果一个规则包含一个 `NOTIFY` 命令， 那么该 `NOTIFY` 命令将被无条件执行，也就是说， 即使规则不施加到任何行上面，该 `NOTIFY` 也会被执行。 比如，在

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

里，一个 `NOTIFY` 事件将在 `UPDATE` 的时候发出，不管是否有满足 `id = 42` 条件的行。 这是一个实现的限制，将来的版本应该修补这个毛病。

兼容性

`CREATE RULE` 是PostgreSQL 语言的扩展，整个规则重写系统都是如此。

又见

ALTER RULE, DROP RULE

CREATE SCHEMA

Name

CREATE SCHEMA -- 定义一个新模式

Synopsis

```
CREATE SCHEMA _schema_name_ [ AUTHORIZATION _user_name_ ] [ _schema_element_ [ ... ] ]
CREATE SCHEMA AUTHORIZATION _user_name_ [ _schema_element_ [ ... ] ]
CREATE SCHEMA IF NOT EXISTS _schema_name_ [ AUTHORIZATION _user_name_ ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION _user_name_
```

描述

`CREATE SCHEMA` 在当前数据库里输入一个新模式。 该模式名将在当前数据库里现存的所有模式名中唯一。

模式实际上是一个名字空间：它包含命名对象(表、数据类型、函数、操作符) 这些名字可以和其它模式里存在的其它对象重名。命名对象要么是通过用模式名作为前缀 "修饰"进行访问，要么是通过设置一个搜索路径包含所需要的模式。 一条带着无修饰对象名的 `CREATE` 命令都是在当前模式中创建创建对象的 (在搜索路径最前面的模式；可以用 `current_schema` 函数来判断)。

另外， `CREATE SCHEMA` 可以包括在新模式中创建对象的子命令。 这些子命令和那些在创建完模式后发出的命令没有任何区别， 只不过是如果使用了 `AUTHORIZATION` 子句，那么所有创建的对象都将被该用户拥有。

参数

`_schema_name_`

要创建的模式名字。如果省略，则使用 `_user_name_` 作为模式名。 这个名字不能以 `pg_` 开头，因为这样的名字保留给系统模式使用。

`_user_name_`

将拥有该模式的用户的角色名。如果省略，缺省为执行该命令的用户名。 要想创建一个属于其他角色的模式，你必须是那个角色的直接或非直接成员，或是超级用户。

`_schema_element_`

一个 SQL 语句，定义一个要在模式里创建的对象。目前，只有 `CREATE TABLE` 、 `CREATE VIEW` 、 `CREATE INDEX` 、 `CREATE SEQUENCE` 、 `CREATE TRIGGER` 和 `GRANT` 是可以接受的子句。 其它类型的对象可以在创建完模式之后的独立命令里创建。

`IF NOT EXISTS`

如果相同名字的模式已经存在，那么什么也不要做（除了发出一个通知）。 当使用此选项时，不能包括 `_schema_element_` 子命令。

注意

要创建模式，调用该命令的用户必需在当前数据库上有 `CREATE` 权限。 （当然，超级用户可以绕开这个检查。）

例子

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户 `joe` 创建模式，模式名也叫 `joe` ：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个名为 `test` 的模式，该模式被用户 `joe` 所拥有， 除非已经有一个名为 `test` 的模式。（这与 `joe` 是否拥有早已存在的模式无关。）

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

创建一个模式并且在里面创建一个表：

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

请注意上面独立的子命令不是由分号结尾的

下面的命令是实现同样结果的等效语句：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

兼容性

SQL 标准允许在 `CREATE SCHEMA` 里面有一个 `DEFAULT CHARACTER SET` 子句以及比目前 PostgreSQL 可以接受的更多的子命令。

SQL 标准声明在 `CREATE SCHEMA` 里的子命令可以以任意顺序出现。 目前PostgreSQL里的实现还不能处理所有子命令里前向引用的情况； 有时候可能需要重排一下子命令的顺序以避免前向引用。

在 SQL 标准里，模式的所有者总是拥有其中的所有对象。 PostgreSQL允许模式包含非模式所有者拥有的对象。 只有在模式所有者将自己模式的 `CREATE` 权限给了其他人， 或者超级用户选择在该模式中创建对象时，才可能出现。

`IF NOT EXISTS` 选项是一个PostgreSQL扩展。

又见

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

Name

CREATE SEQUENCE -- 定义一个新序列发生器

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE _name_ [ INCREMENT [ BY ] _increment_ ]
    [ MINVALUE _minvalue_ | NO MINVALUE ] [ MAXVALUE _maxvalue_ | NO MAXVALUE ]
    [ START [ WITH ] _start_ ] [ CACHE _cache_ ] [ [ NO ] CYCLE ]
    [ OWNED BY { _table_name_._column_name_ | NONE } ]
```

描述

`CREATE SEQUENCE` 将向当前数据库里增加一个新的序列号生成器。 包括创建和初始化一个新的名为 `_name_` 的单行表。生成器将被使用此命令的用户所有。

如果给出了一个模式名，那么该序列就在给定的模式中创建的。 否则它会在当前模式中创建。临时序列存在于一个特殊的模式中， 因此创建临时序列的时候不能给出模式名。序列名必需和同一模式中的其它序列、 表、索引、视图或外表的名字不同。

在创建序列后，你可以使用 `nextval` 、 `currval` 和 `setval` 函数操作序列。 这些函数在Section 9.16中有详细文档。

尽管你不能直接更新一个序列，但你可以使用：

```
SELECT * FROM _name_;
```

检查一个序列的参数和当前状态。特别是序列的 `last_value` 字段显示了任意会话最后分配的数值。 （当然，如果其它会话正积极地使用 `nextval` ，这些值在被打印出来的时候可能就已经过时了。）

参数

`TEMPORARY` 或 `TEMP`

如果声明了这个修饰词，那么该序列对象只为这个会话创建， 并且在会话结束的时候自动删除。在临时序列存在的时候， 除非用模式修饰的名字引用，否则同名永久序列是不可见的(在同一会话里)。

`_name_`

将要创建的序列名(可以用模式修饰)

`_increment_`

可选子句 `INCREMENT BY` `_increment_` 指定序列的步长。 一个正数将生成一个递增的序列，一个负数将生成一个递减的序列。缺省值是 1 。

`_minvalue_``NO MINVALUE`

可选的子句 `MINVALUE` `_minvalue_` 指定序列的最小值。如果没有声明这个子句或者声明了 `NO MINVALUE` ， 那么递增序列的缺省为 1 ， 递减序列的缺省为-2⁶³-1。

`_maxvalue_` `NO MAXVALUE`

可选的子句 `MAXVALUE` `_maxvalue_` 指定序列的最大值。如果没有声明这个子句或者声明了 `NO MAXVALUE` ， 那么递增序列的缺省为2⁶³-1，递减序列的缺省为 -1 。

`_start_`

可选的子句 `START WITH` `_start_` 指定序列的起点。缺省初始值对于递增序列为 `_minvalue_` ， 对于递减序列为 `_maxvalue_` 。

`_cache_`

可选的子句 `CACHE` `_cache_` 为快速访问而在内存里预先存储多少个序列号。最小值(也是缺省值)是 1 ， 表示一次只能生成一个值，也就是说没有缓存。

`CYCLE` `NO CYCLE`

`CYCLE` 选项可用于使序列到达 `_maxvalue_` 或 `_minvalue_` 时可循环并继续下去。 也就是如果达到极限，生成的下一个数据将分别是 `_minvalue_` 或 `_maxvalue_` 。

如果声明了 `NO CYCLE` ，那么在序列达到其最大值之后任何对 `nextval` 的调用都将返回一个错误。 如果既没有声明 `CYCLE` 也没有声明 `NO CYCLE` ， 那么 `NO CYCLE` 是缺省。

`OWNED BY` `_table_name_` . `_column_name_` `OWNED BY NONE`

`OWNED BY` 选项将序列关联到一个特定的表字段上。这样， 在删除那个字段或其所在表的时候将自动删除绑定的序列。 指定的表和序列必须被同一个用户所拥有，并且在在同一个模式中。 默认的 `OWNED BY NONE` 表示不存在这样的关联。

注意

使用 `DROP SEQUENCE` 删除一个序列。

序列是基于 `bigint` 运算的，因此其范围不能超过八字节的整数范围(-9223372036854775808 到 9223372036854775807)。 在一些古老的平台上可能没有对八字节整数的编译器支持， 这种情况下序列使用普通的 `integer` 运算范围(-2147483648 到 +2147483647)。

如果 `_cache_` 大于一， 并且这个序列对象将被用于多会话并发的场合，那么可能会有不可预料的结果发生。 每个会话在每次访问序列对象的过程中都将分配并缓存随后的序列值， 并且相应增加序列对象的 `last_value` 。这样， 同一个事务中的随后的 `_cache_` -1 次 `nextval` 将只是返回预先分配的数值，而不是使用序列对象。 因此，任何在会话中分配了却没有使用的数字都将在会话结束时丢失， 从而导致序列里面出现"空洞"。

另外，尽管系统保证为多个会话分配独立的序列值，但是如果考虑所有会话， 那么这个数值可能会丢失顺序。比如，如果 `_cache_` 为 10 ，那么会话 A 保留了 1..10 并且返回 `nextval` =1， 然后会话 B 可能会保留 11..20 然后在会话 A 生成 `nextval` =2 之前返回 `nextval` =11。因此， 对于 `_cache_` 等于一的情况， 可以安全地假设 `nextval` 值是顺序生成的；而如果把 `_cache_` 设置为大于一， 那么你只能假设 `nextval` 值总是不同的，却不按顺序生成。 同样， `last_value` 将反映任何会话保留的最后数值， 不管它是否曾被 `nextval` 返回。

另外一个考虑是在这样的序列上执行的 `setval` 将不会被其它会话注意到，直到它们用光他们自己缓存的数值。

例子

创建一个叫 `serial` 的递增序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从此序列中选出下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     101
```

从此序列中选出下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     102
```

在一个 `INSERT` 中使用此序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一个 `COPY FROM` 后更新序列：

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

兼容性

`CREATE SEQUENCE` 遵循SQL标准， 只有下面的例外：

- 还不支持标准的 `AS <数据类型>` 表达式。
- 使用 `nextval()` 函数而不是标准的 `NEXT VALUE FOR` 表达式获取下一个数值。
- `OWNED BY` 子句是PostgreSQL的扩展。

又见

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

Name

CREATE SERVER -- 定义一个新的外服务器

Synopsis

```
CREATE SERVER _server_name_ [ TYPE '_server_type_' ] [ VERSION '_server_version_' ]
    FOREIGN DATA WRAPPER _fdw_name_
    [ OPTIONS ( _option_ '_value_' [, ... ] ) ]
```

描述

`CREATE SERVER` 定义一个新的外部服务器。定义该服务器的用户将成为其所有者。

一个外部服务器通常将关于访问外部数据源的连接信息的外部数据容器封装起来。 用户特定的额外信息可以由用户映射的方式来指定。 额外的用户特定的连接信息可以由用户映射的方式来指定。

服务器名称必须是数据库内唯一的。

创建服务器需要外部数据容器上有 `USAGE` 权限。

参数

`_server_name_`

创建外部服务器的名字。

`_server_type_`

可选的服务器类型，对外部数据容器可能有用。

`_server_version_`

可选的服务器版本，对外部数据容器可能有用。

`_fdw_name_`

管理服务器的外部数据容器的名字。

`OPTIONS (` `_option_` ' `_value_` '[, ...])

该子句指定了服务器的选项。该选项通常定义了连接的详细信息， 但实际上名称和值是依赖服务器的外部数据容器。

注意

当使用 dblink 模块的时候, 一个外部服务器的名字可以使用如dblink_connect函数， 用于表示连接参数。这样需要在外部服务器上有 `USAGE` 的权限， 才能够这样使用。

示例

通过使用外部数据容器 `postgres_fdw` 创建外部服务器 `myserver` :

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname 'foo
```

参见 postgres_fdw了解更多详情。

兼容性

`CREATE SERVER` 符合 ISO/IEC 9075-9 (SQL/MED)标准。

另请参见

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE TABLE

Name

CREATE TABLE -- 定义一个新表

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] _ta
  { _column_name_ _data_type_ [ COLLATE _collation_ ] [ _column_constraint_ [ ... ] ]
    | _table_constraint_
    | LIKE _source_table_ [ _like_option_ ... ] }
    [, ... ]
] )
[ INHERITS ( _parent_table_ [, ... ] ) ]
[ WITH ( _storage_parameter_ [= _value_] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE _tablespace_name_ ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] _ta
    OF _type_name_ [ (
  { _column_name_ WITH OPTIONS [ _column_constraint_ [ ... ] ]
    | _table_constraint_ }
    [, ... ]
) ]
[ WITH ( _storage_parameter_ [= _value_] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE _tablespace_name_ ]

 这里的`_column_constraint_`是:

[ CONSTRAINT _constraint_name_ ]
{ NOT NULL |
  NULL |
  CHECK ( _expression_ ) [ NO INHERIT ] |
  DEFAULT _default_expr_ |
  UNIQUE _index_parameters_ |
  PRIMARY KEY _index_parameters_ |
  REFERENCES _reftable_ [ ( _refcolumn_ ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE _action_ ] [ ON UPDATE _action_ ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

 而`_table_constraint_`是:

[ CONSTRAINT _constraint_name_ ]
{ CHECK ( _expression_ ) [ NO INHERIT ] |
  UNIQUE ( _column_name_ [, ... ] ) _index_parameters_ |
  PRIMARY KEY ( _column_name_ [, ... ] ) _index_parameters_ |
  EXCLUDE [ USING _index_method_ ] ( _exclude_element_ WITH _operator_ [, ... ] ) _index_
  FOREIGN KEY ( _column_name_ [, ... ] ) REFERENCES _reftable_ [ ( _refcolumn_ [, ... ] )
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE _action_ ] [ ON UPDATE _act
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

 `_like_option_`是:

{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }

 在`UNIQUE`、`PRIMARY KEY`和
 `EXCLUDE`约束中的`_index_parameters_`是:

[ WITH ( _storage_parameter_ [= _value_] [, ... ] ) ]
[ USING INDEX TABLESPACE _tablespace_name_ ]

`_exclude_element_` in an `EXCLUDE` constraint is:

{ _column_name_ | ( _expression_ ) } [ _opclass_ ] [ ASC | DESC ] [ NULLS { FIRST | LAST
```

描述

`CREATE TABLE` 在当前数据库创建一个新的空白表。 该表将由发出此命令的用户所拥有。

如果给出了模式名(比如 `CREATE TABLE myschema.mytable ...`)， 那么在指定的模式中创建表，否则在当前模式中创建。临时表存在于一个特殊的模式里， 因此创建临时表的时候不能指定模式名。表名字必需在同一模式中的其它表、 序列、索引、视图或外部表名字中唯一。

`CREATE TABLE` 还自动创建一个与该表的行对应的复合数据类型。 因此，表不能和同模式中的现有数据类型同名。

可选的约束子句声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。 约束是一个 SQL 对象，它以多种方式协助在表上定义有效数值的集合。

定义约束有两种方法：表约束和列约束。列约束是作为一个列定义的一部分定义的。 而表约束并不和某个列绑在一起，它可以作用于多个列上。每个列约束也可以写成表约束； 如果某个约束只影响一个列，那么列约束只是符号上的简洁方式而已。

要能创建一个表，你必须分别在所有列类型或 `OF` 子句的类型上有 `USAGE` 权限。

参数

`TEMPORARY` 或 `TEMP`

如果声明了，则创建为临时表。临时表在会话结束或(可选)当前事务的结尾 (参阅下面的 `ON COMMIT`)自动删除。除非用模式修饰的名字引用， 否则现有的同名永久表在临时表存在期间，在本会话过程中是不可见的。 任何在临时表上创建的索引也都会被自动删除。

自动清理进程不能访问，因此不能清理或分析临时表。 由于这个原因，适当的清理和分析操作应该通过会话SQL命令来执行。例如， 如果一个临时表用于复杂查询中，那么在填充临时表之后在临时表上运行 `ANALYZE` 是明智的。

可以选择在 `TEMPORARY` 或 `TEMP` 前面放上 `GLOBAL` 或 `LOCAL` 。不过这目前对PostgreSQL来说没有任何区别， 并且已经废弃了；可以参阅兼容性。

`UNLOGGED`

如果指定了，则表作为非日志表来创建。写入到非日志表中的数据并不写入预写式日志 （参阅Chapter 29）， 这使得他们比普通表快的多。不过， 他们不是崩溃安全的：一个非日志表在崩溃或不清理关机之后被自动截断。 非日志表的内容也不复制到备用服务器。任何在非日志表上创建的索引也自动是非日志的。

`IF NOT EXISTS`

如果同名的关系已经存在，那么不要抛出一个错误。在这种情况下发出一个通知。 请注意，不保证已经存在的关系和要创建的关系相像。

`_table_name_`

要创建的表的名字(可以用模式修饰)

`OF` `_type_name_`

创建一个类型化的表，它的结构来自指定的复合类型（名字可以有模式修饰）。一个类型化的表绑定到它的类型；例如，如果类型被删除（使用 `DROP TYPE ... CASCADE`）， 则该表也将被删除。

当类型化的表被创建时，字段的数据类型取决于底层复合类型， 而不是通过 `CREATE TABLE` 命令指定。 但是 `CREATE TABLE` 命令可以给表添加缺省和约束， 并且可以指定存储参数。

`_column_name_`

在新表中要创建的字段名字。

`_data_type_`

该字段的数据类型。它可以包括数组说明符。有关PostgreSQL 支持的数据类型的更多信息，请参考Chapter 8。

`COLLATE` `_collation_`

`COLLATE` 给字段（必须是可排序的数据类型）分配一个排序规则。 如果没有指定，则使用字段数据类型的缺省排序。

`INHERITS (` `_parent_table_` [, ...])

可选的 `INHERITS` 子句声明一系列的表，新表自动从这一系列表中继承所有字段。

使用 `INHERITS` 将在新的子表和其父表之间创建一个永久的关系。 对父表结构的修改通常也会传播到子表。缺省时，扫描父表的时候也会扫描子表。

如果在多个父表中存在同名字段，那么就会报告一个错误， 除非这些字段的数据类型在每个父表里都是匹配的。如果没有冲突， 那么重复的字段在新表中融合成一个字段。如果列出的新表字段名和继承字段同名， 那么它的数据类型也必须和继承字段匹配，并且这些字段定义会融合成一个。 如果新表为该字段明确声明了缺省值，那么此缺省值将覆盖任何继承字段的缺省值。 否则，该字段的所有父字段缺省值都必须相同，否则就会报错。

`CHECK` 约束本质上以和字段一样的方式合并：如果多个父表和/或新表的定义包含相同名字的 `CHECK` 约束，那么这些约束必须都有相同的检查表达式，否则会报告一个错误。 有相同名字和表达式的约束将合并到一个。在父表中标记为 `NO INHERIT` 的约束将不被考虑。请注意，新表中未命名的 `CHECK` 约束将永远不被合并， 因为它将永远都有一个唯一的名字。

字段 `STORAGE` 设置也从父表中拷贝。

`LIKE` `_source_table_` [`_like_option_` ...]

`LIKE` 子句声明一个表， 新表自动从这个表里面继承所有字段名及其数据类型和非空约束。

和 `INHERITS` 不同，新表与原来的表之间在创建动作完毕之后是完全无关的。 在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

字段缺省表达式只有在声明了 `INCLUDING DEFAULTS` 之后才会包含进来。 缺省是不包含缺省表达式的，结果是新表中所有字段的缺省值都是 NULL 。

非空约束将总是复制到新表中， `CHECK` 约束则仅在指定了 `INCLUDING CONSTRAINTS` 的时候才复制。源表上的索引、 `PRIMARY KEY` 和 `UNIQUE` 约束仅在指定了 `INCLUDING INDEXES` 子句的时候才在新表上创建。 此规则同时适用于表约束和列约束。

拷贝字段定义的 `STORAGE` 设置只在声明了 `INCLUDING STORAGE` 的时候才拷贝。缺省是不包括 `STORAGE` 设置的， 结果是新表中的拷贝字段有特定类型的缺省设置。关于 `STORAGE` 设置的更多信息，请参阅Section 58.2。

拷贝字段、约束和索引的注释只在声明了 `INCLUDING COMMENTS` 的时候拷贝。缺省是不包含注释的，结果是新表中拷贝的字段和约束没有注释。

`INCLUDING ALL` 是 `INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STOR` 的一个简写形式。

请注意，和 `INHERITS` 不同，通过 `LIKE` 拷贝的列和约束并不使用相同的名字进行融合。 如果明确的指定了相同的名字或者在另外一个 `LIKE` 子句中，那么将会报错。

`LIKE` 子句也可以用来从视图、外表或复合类型中拷贝字段。 不适用的选项（如，从一个视图中 `INCLUDING INDEXES` ）都被忽略。

`CONSTRAINT` `_constraint_name_`

可选的列约束或表约束的名字。如果约束本身是非法的，那么其名字将会出现在错误信息中， 因此像 `col must be positive` 这样的名字可以表达有用的约束信息。 如果约束名中含有空格则必须用双引号界定。如果没有指定约束名，那么系统将会自动生成一个名字。

`NOT NULL`

字段不允许包含 NULL 值

`NULL`

字段允许包含 NULL 值，这是缺省。

这个子句的存在只是为和那些非标准 SQL 数据库兼容。不建议在新应用中使用它。

`CHECK (` `_expression_`) [NO INHERIT]

`CHECK` 约束声明一个生成布尔结果的表达式， 每次将要插入的新行或者将要被更新的行必须使表达式结果为真或未知才能成功， 否则会抛出一个异常并且不会修改数据库。 声明为字段约束的检查约束应该只引用该字段的数值， 而在表约束里出现的表达式可以引用多个字段。

目前， `CHECK` 表达式不能包含子查询也不能引用除当前行字段之外的变量。

用 `NO INHERIT` 标记的约束将不会传递到子表中去。

`DEFAULT` `_default_expr_`

`DEFAULT` 子句给字段指定缺省值。该数值可以是任何不含变量的表达式 (不允许使用子查询和对本表中的其它字段的交叉引用)。 缺省表达式的数据类型必须和字段类型匹配。

缺省表达式将被用于任何未声明该字段数值的插入操作。 如果没有指定缺省值则缺省值为NULL 。

`UNIQUE` (column constraint) `UNIQUE (` `_column_name_` [, ...]) (table constraint)

`UNIQUE` 约束表示表里的一个或多个字段的组合必须在全表范围内唯一。 唯一约束的行为和列约束一样，只不过多了跨多行的能力。

对于唯一约束而言，NULL 被认为是互不相等的。

每个唯一约束都必须给其使用的字段集合赋予一个与其它唯一约束都不同的名字， 并且也不能和主键约束的名字相同，否则就被认为是同样的约束写了两次。

`PRIMARY KEY` (column constraint) `PRIMARY KEY (` `_column_name_` [, ...]) (table constraint)

主键约束表明表中的一个或者一些字段只能包含唯一(不重复)的非 NULL 值。 从技术上讲， `PRIMARY KEY` 只是 `UNIQUE` 和 `NOT NULL` 的组合，不过把一套字段标识为主键同时也体现了模式设计的元数据， 因为主键意味着可以拿这套字段用做行的唯一标识。

一个表只能声明一个主键，不管是作为字段约束还是表约束。

主键约束使用的字段集合应该与其它唯一约束都不同。

`EXCLUDE [ USING` `_index_method_`] (`_exclude_element_` WITH `_operator_` [, ...]) `_index_parameters_` [WHERE (`_predicate_`)]

`EXCLUDE` 子句定义一个排除约束，保证了两个行在指定的字段或使用指定操作符的表达式上比较时， 并不都返回 `TRUE` 。如果所有指定的操作符测试都相等， 那么就等同于 `UNIQUE` 约束，尽管一个普通的唯一约束会更快速。 不过，排除约束可以指定比简单相等更普通的约束。例如， 你可以通过使用 `&&` 操作符指定一个约束， 在一个表中没有两个行包含重叠的圆圈(参阅Section 8.8)。

排除约束是使用索引实现的，所以每个指定的操作符都必须和索引访问方法的一个合适的操作符类 （参阅Section 11.9）相关。要求操作符是可交换的。 每个 `_exclude_element_` 可以指定一个操作符类和/或排序选项；这些在CREATE INDEX中有充分的描述。

访问方法必须支持 `amgettuple` (参阅Chapter 54)； 目前，这意味着不能使用GIN。尽管允许使用， 但是在排除约束中使用B-tree或哈希索引没什么意义， 因为一个普通唯一约束做的更好。所以实际中，访问方法总是GiST 或SP-GiST。

`_predicate_` 允许在表的子集上声明一个排除约束； 在内部这创建了一个部分索引。请注意，谓语需要有括号包围。

`REFERENCES` `_reftable_` [(`_refcolumn_`)] [MATCH `_matchtype_`] [ON DELETE `_action_`] [ON UPDATE `_action_`] (column constraint) `FOREIGN KEY (` `_column_name_` [, ...]) REFERENCES `_reftable_` [(`_refcolumn_` [, ...])] [MATCH `_matchtype_`] [ON DELETE `_action_`] [ON UPDATE `_action_`] (table constraint)

这些子句声明一个外键约束，外键约束要求新表中一列或多列组成的组应该只包含 匹配被参考的表中对应字段中的值。如果省略 `_refcolumn_` ，则使用 `_reftable_` 的主键。 被参考字段必须是被参考表中的唯一字段或者主键。请注意， 不能在临时表和永久表之间定义外键约束。

向参考字段插入的数值将使用给出的匹配类型与被参考表中被参考列的数值进行匹配。 有三种匹配类型： `MATCH FULL` , `MATCH PARTIAL` , `MATCH SIMPLE` (缺省)。 `MATCH FULL` 不允许一个多字段外键的字段为 NULL ， 除非所有外键字段都为 NULL ； 如果都是NULL，那么该行在引用的表中不需要有匹配。 `MATCH SIMPLE` 允许任意外键字段为 NULL ；如果都是NULL，那么该行在引用的表中不需要有匹配。 `MATCH PARTIAL` 目前尚未实现。 （当然， `NOT NULL` 约束可以应用于引用字段，以阻止这些情况的发生。）

另外，当被参考字段中的数据改变的时候，那么将对本表的字段中的数据执行某种操作。 `ON DELETE` 子句声明当被参考表中的被参考行被删除的时候要执行的操作。 类似的， `ON UPDATE` 子句声明被参考表中被参考字段更新为新值的时候要执行的动作。 如果该行被更新，但被参考的字段实际上没有变化，那么就不会有任何动作。 除了 `NO ACTION` 检查之外的其他参考动作都不能推迟， 即使该约束声明为可推迟也是如此。下面是每个子句的可能动作：

`NO ACTION`

生成一个错误，表明删除或更新将产生一个违反外键约束的动作。 如果该约束是可推迟的，并且如果还存在任何引用行，那么这个错误将在检查约束的时候生成。 这是缺省动作。

`RESTRICT`

生成一个表明删除或更新将导致违反外键约束的错误。 和 `NO ACTION` 一样，只是动作不可推迟。

`CASCADE`

删除任何引用了被删除行的行，或者分别把引用行的字段值更新为被参考字段的新数值。

`SET NULL`

把引用行设置为 NULL 。

`SET DEFAULT`

把引用字段设置为它们的缺省值。（如果他们为非空，那么被引用的表中必须有一行匹配缺省值， 否则该操作将会失败。）

如果被参考字段经常更新，那么给引用字段增加一个索引可能是合适的， 这样与外键约束相关联的引用动作可以更有效地执行。

`DEFERRABLE` `NOT DEFERRABLE`

这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。 可推迟约束可以推迟到事务结尾使用SET CONSTRAINTS命令检查。 缺省是 `NOT DEFERRABLE` 。目前，只有 `UNIQUE` 、 `PRIMARY KEY` 、 `EXCLUDE` 和 `REFERENCES` （外键） 约束接受这个子句。 `NOT NULL` 和 `CHECK` 约束都是不可推迟的。

`INITIALLY IMMEDIATE` `INITIALLY DEFERRED`

如果约束是可推迟的，那么这个子句声明检查约束的缺省时间。 如果约束是 `INITIALLY IMMEDIATE` (缺省)， 那么每条语句之后就立即检查它。如果约束是 `INITIALLY DEFERRED` ， 那么只有在事务结尾才检查它。约束检查的时间可以用 SET CONSTRAINTS命令修改。

`WITH (` `_storage_parameter_` [= `_value_`] [, ...])

这个子句为表或索引指定一个可选的存储参数，参见存储参数获取更多信息。 用于表的 `WITH` 子句还可以包含 `OIDS=TRUE` 或单独的 `OIDS` 来指定给新表中的每一行都分配一个OID(对象标识符)， 或者 `OIDS=FALSE` 表示不分配 OID 。如果没有指定 `OIDS` 默认行为取决于default_with_oids配置参数。 如果新表是从有 OID 的表继承而来，那么即使明确指定 `OIDS=FALSE` 也将强制按照 `OIDS=TRUE` 执行。

如果明确或隐含的指定了 `OIDS=FALSE` ，新表将不会存储 OID， 也不会为插入的行分配 OID 。这将减小 OID 的开销并因此延缓了 32-bit OID 计数器的循环。因为一旦计数器发生循环之后 OID 将不能被视为唯一， 这将大大降低 OID 的实用性。另外， 排除了 OID 的表也为每条记录减小了 4 字节的存储空间（在大多数机器上）， 从而可以稍微提升一些性能。

可以使用ALTER TABLE从已有的表中删除 OID 列。

`WITH OIDS` `WITHOUT OIDS`

这些是被废弃的、分别等价于 `WITH (OIDS)` 和 `WITH (OIDS=FALSE)` 的语法。如果想同时给出 `OIDS` 设置和存储参数，必须使用 `WITH ( ... )` 语法；见上文。

`ON COMMIT`

可以使用 `ON COMMIT` 控制临时表在事务块结尾的行为。 这三个选项是：

`PRESERVE ROWS`

在事务的结尾不采取任何特别的动作，这是缺省。

`DELETE ROWS`

在每个事务块的结尾都删除临时表中的所有行。 本质上是在每次提交事务后自动执行一个TRUNCATE命令。

`DROP`

在当前事务块的结尾删除临时表。

`TABLESPACE` `_tablespace_name_`

指定新表将要在 `_tablespace_name_` 表空间内创建。如果没有声明，将使用default_tablespace， 如果该表为临时表，那么将使用temp_tablespaces。

`USING INDEX TABLESPACE` `_tablespace_name_`

这个子句允许选择与一个 `UNIQUE` 、 `PRIMARY KEY` 或 `EXCLUDE` 约束相关的索引创建时所在的表空间。如果没有声明， 将使用default_tablespace， 如果该表为临时表，那么将使用temp_tablespaces。

存储参数

`WITH` 子句可以为表指定存储参数，并在索引上创建 `UNIQUE` 、 `PRIMARY KEY` 或 `EXCLUDE` 约束。用于索引的存储参数在CREATE INDEX里面描述。 目前可以在表上使用的存储参数在下面列出。对于每个参数，除非指明了， 有一个额外的参数是相同的名字加上 `toast.` 前缀，可以用来控制表的次要TOAST表的行为，如果有 (参阅Section 58.2获取更多关于TOAST的信息)。 请注意，TOAST表从它的父表中继承 `autovacuum_*` 值， 如果没有设置 `toast.autovacuum_*` 设置。

`fillfactor` (`integer`)

一个表的填充因子(fillfactor)是一个介于 10 和 100 之间的百分数。 100(完全填充)是默认值。如果指定了较小的填充因子， `INSERT` 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行， 这就使得 `UPDATE` 有机会在同一页上放置同一条记录的新版本， 这比把新版本放置在其它页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，较小的填充因子则更加有效。 这个参数不能为TOAST表设置。

`autovacuum_enabled` , `toast.autovacuum_enabled` (`boolean`)

在一个特别的表上启用或禁用自动清理守护进程。如果为真， 当更新或删除的元组的数量超过 `autovacuum_vacuum_threshold` 加上 `autovacuum_vacuum_scale_factor` 倍的当前关系中评估的活动的元组时， 自动清理守护进程将在一个特定的表上发起一个 `VACUUM` 操作。 相似的，当插入、更新或删除的元组的数量超过 `autovacuum_analyze_threshold` 加上 `autovacuum_analyze_scale_factor` 倍的当前关系中评估的活动的元组时， 自动清理守护进程将发起一个 `ANALYZE` 操作。如果为假， 这个表将不被自动清理，除了防止事务Id循环。参阅Section 23.1.5 获取更多关于预防循环的信息。注意，这个变量从autovacuum 设置中继承值。

`autovacuum_vacuum_threshold` , `toast.autovacuum_vacuum_threshold` (`integer`)

在一个特定的表上发起 `VACUUM` 操作之前，更新或删除的元组的最小值。

`autovacuum_vacuum_scale_factor` , `toast.autovacuum_vacuum_scale_factor` (`float4`)

添加到 `autovacuum_vacuum_threshold` 的 `reltuples` 的乘数。

`autovacuum_analyze_threshold` (`integer`)

在一个特定的表上发起 `ANALYZE` 操作之前，插入、更新或删除的元组的最小值。

`autovacuum_analyze_scale_factor` (`float4`)

添加到 `autovacuum_analyze_threshold` 的 `reltuples` 乘数。

`autovacuum_vacuum_cost_delay` , `toast.autovacuum_vacuum_cost_delay` (`integer`)

自定义autovacuum_vacuum_cost_delay参数。

`autovacuum_vacuum_cost_limit` , `toast.autovacuum_vacuum_cost_limit` (`integer`)

自定义autovacuum_vacuum_cost_limit参数。

`autovacuum_freeze_min_age` , `toast.autovacuum_freeze_min_age` (`integer`)

自定义autovacuum_vacuum_cost_limit参数。请注意， 自动清理将忽略设置一个每表 `autovacuum_freeze_min_age` 大于半个系统范围的autovacuum_freeze_max_age设置的尝试。

`autovacuum_freeze_max_age` , `toast.autovacuum_freeze_max_age` (`integer`)

自定义autovacuum_freeze_max_age参数。请注意， 自动清理将忽略设置一个每表 `autovacuum_freeze_max_age` 大于系统范围的设置的尝试（它只能设置的较小一些）。注意， 你可以将 `autovacuum_freeze_max_age` 设置的非常小，甚至为零， 这通常是不明智的，因为它将强制频繁的清理。

`autovacuum_freeze_table_age` , `toast.autovacuum_freeze_table_age` (`integer`)

自定义vacuum_freeze_table_age参数。

注意

不建议在新应用中使用 OID ， 可能情况下，更好的选择是使用一个 `SERIAL` 或者其它序列发生器做表的主键。如果一个应用使用了 OID 标识表中的特定行， 那么建议在该表的 `oid` 字段上创建一个唯一约束， 以确保该表的 OID 即使在计数器循环之后也是唯一的。 如果你需要一个整个数据库范围的唯一标识，那么就要避免假设 OID 是跨表唯一的， 你可以用 `tableoid` 和行 OID 的组合来实现这个目的。

> **Tip:** 对那些没有主键的表，不建议使用 `OIDS=FALSE`， 因为如果既没有 OID 又没有唯一数据字段，那么就很难标识特定的行。

PostgreSQL 自动为每个唯一约束和主键约束创建一个索引以确保其唯一性。因此， 不必为主键字段明确的创建索引。参阅CREATE INDEX获取更多信息。

唯一约束和主键在目前的实现里是不能继承的。 如果把继承和唯一约束组合在一起会导致无法运转。

一个表不能超过 1600 个字段。实际的限制比这个更低，因为还有元组长度限制。

例子

创建 `films` 和 `distributors` 表：

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建一个带有 2 维数组的表：

```
CREATE TABLE array_int (
    vector  int[][]
);
```

为表 `films` 定义一个唯一表约束。 唯一表约束可以在表的一个或多个字段上定义：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个检查列约束：

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

定义一个检查表约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40)
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

为表 `films` 定义一个主键表约束。

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

为表 `distributors` 定义一个主键约束。下面两个例子是等效的， 第一个例子使用了表约束语法，第二个使用了列约束语法。

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

下面这个例子给字段 `name` 赋予了一个文本常量缺省值， 并且将字段 `did` 的缺省值安排为通过选择序列对象的下一个值生成。 `modtime` 的缺省值将是该行插入的时间戳。

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表 `distributors` 上定义两个 `NOT NULL` 列约束， 其中之一明确给出了名字：

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

为 `name` 字段定义一个唯一约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

相同的，声明为一个表约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

创建同样的表，并为表以及唯一索引指定 70% 填充率：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

创建一个带有排除约束的表 `circles`， 阻止任意两个圆重叠：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

在表空间 `diskvol1` 里创建 `cinemas` 表：

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

创建一个复合类型和类型化的表：

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

兼容性

`CREATE TABLE` 遵循SQL标准， 一些例外情况在下面列出。

临时表

尽管 `CREATE TEMPORARY TABLE` 的语法和 SQL 标准的类似，但是效果是不同的。 在标准里，临时表只是定义一次并且从空内容开始自动存在于任何需要它们的会话中。 PostgreSQL要求每个会话为它们使用的每个临时表发出它们自己的 `CREATE TEMPORARY TABLE` 命令。 这样就允许不同的会话将相同的临时表名字用于不同的目的， 而标准的实现方法则把一个临时表名字约束为具有相同的表结构。

标准定义的临时表的行为被广泛地忽略了。PostgreSQL 在这方面上的行为类似于许多其它 SQL 数据库系统。

SQL标准也区分全局和局部临时表，局部临时表对于每个会话中的每个SQL模块都有一个单独的内容设置， 虽然它的定义仍然在会话中共享。因为PostgreSQL 不支持SQL模块，所以这个差别在PostgreSQL不相关。

出于兼容考虑，PostgreSQL将接受临时表声明中的 `GLOBAL` 和 `LOCAL` 关键字，但是他们没有任何作用。 不建议使用这些关键字，因为PostgreSQL 的未来版本可能采取更加标准兼容的它们的含义。

临时表的 `ON COMMIT` 子句也类似于 SQL 标准，但是有些区别。 如果忽略了 `ON COMMIT` 子句，SQL 标准声明缺省的行为是 `ON COMMIT DELETE ROWS` 。但是PostgreSQL 里的缺省行为是 `ON COMMIT PRESERVE ROWS` 。 在 SQL 标准里不存在 `ON COMMIT DROP` 选项。

非延迟的唯一性约束

当 `UNIQUE` 或 `PRIMARY KEY` 约束是不可延迟的时， 当插入或修改一个行时，PostgreSQL 立即检查唯一性。SQL标准说唯一约束应该只在语句的结尾强制执行； 这使得它和PostgreSQL不同，例如， 一个命令更新多个键值。要获得标准兼容的行为，声明该约束为 `DEFERRABLE` 但是不能延迟（也就是说， `INITIALLY IMMEDIATE` ）。 要知道，这会比立即检查唯一性要慢的多。

列检查约束

SQL 标准说 `CHECK` 列约束只能引用他们作用的字段； 只有 `CHECK` 表约束才能引用多个字段。 PostgreSQL 并不强制这个限制；它把字段和表约束看作相同的东西。

`EXCLUDE` 约束

`EXCLUDE` 约束类型是一个PostgreSQL扩展。

`NULL` "约束"

`NULL` "约束"(实际上不是约束)是 PostgreSQL对 SQL 标准的扩展， 包括它是为了和其它一些数据库系统兼容以及为了和 `NOT NULL` 约束对称。因为它是任何字段的缺省，所以它的出现是没有意义的。

继承

通过 `INHERITS` 子句的多重继承是PostgreSQL 语言的扩展。SQL:1999 及以后的标准使用不同的语法和语义定义了单继承。 SQL:1999风格的继承还没有在PostgreSQL中实现。

零字段表

PostgreSQL允许创建没有字段的表(比如 `CREATE TABLE foo();`)。这是对 SQL 标准的扩展，标准不允许存在零字段表。零字段表本身没什么用， 但是禁止他们会给 `ALTER TABLE DROP COLUMN` 带来很奇怪的情况， 所以，这个时候忽视标准的限制概念非常清楚。

`WITH` 子句

`WITH` 子句是PostgreSQL的扩展，同样， 存储参数和 OID 也是扩展。

表空间

PostgreSQL的表空间概念不是标准的东西。 因此 `TABLESPACE` 和 `USING INDEX TABLESPACE` 都是扩展。

类型化的表

类型化的表实现了SQL标准的一个子集。根据标准， 类型化的表有对应于底层复合类型和一个其他的"自我参考字段"。 PostgreSQL没有明确支持这些自我参考字段，但是相同的效果可以使用OID特性达到。

又见

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

Name

CREATE TABLE AS -- 从一条查询的结果中定义一个新表

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE _table_name_
    [ (_column_name_ [, ...] ) ]
    [ WITH ( _storage_parameter_ [= _value_] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE _tablespace_name_ ]
    AS _query_
    [ WITH [ NO ] DATA ]
```

描述

`CREATE TABLE AS` 创建一个表并且用来自 `SELECT` 命令的结果填充该表。该表的字段和 `SELECT` 输出字段的名字及数据类型相关。 （不过你可以通过明确地给出一个字段名字列表来覆盖 SELECT 输出字段的名字。）

`CREATE TABLE AS` 和创建视图有点像， 不过两者之间实在是差异很大：它创建一个新表并且只对查询计算一次来填充这个新表。 新表不能跟踪源表的变化。相比之下，每次做查询的时候， 视图都重新计算定义它的 `SELECT` 语句。

参数

`GLOBAL` 或 `LOCAL`

忽略，仅仅为了兼容性而存在。这些关键字的使用已经废弃了； 请参考CREATE TABLE获取细节。

`TEMPORARY` 或 `TEMP`

如果声明了这个选项，则该表作为临时表创建。 参阅CREATE TABLE获取细节。

`UNLOGGED`

如果声明了这个选项，则该表作为非日志表创建。 参阅CREATE TABLE获取细节。

`_table_name_`

要创建的表名(可以用模式修饰)。

`_column_name_`

新表中字段的名称。如果没有提供字段名字，那么就从查询的输出字段名中获取。

`WITH (` `_storage_parameter_` [= `_value_`] [, ...])

这个子句为新表指定了可选的存储参数；参见存储参数获取更多信息。 `WITH` 子句还可以包含 `OIDS=TRUE` （或只是 `OIDS` ） 来为新表中的行分配和存储 OID(对象表示符)；或者用 `OIDS=FALSE` 表示不分配 OID 。参见CREATE TABLE获取更多信息。

`WITH OIDS``WITHOUT OIDS`

这些是反对使用的、分别等价于 `WITH (OIDS)` 和 `WITH (OIDS=FALSE)` 的语法。如果你希望同时给出 `OIDS` 设置和存储参数，必须使用 `WITH ( ... )` 语法；见下文。

`ON COMMIT`

可以使用 `ON COMMIT` 控制临时表在事务块结尾的行为。 三个选项是：

`PRESERVE ROWS`

在事务的结尾不采取任何特别的动作，这是缺省。

`DELETE ROWS`

在每个事务块的结尾都删除临时表中的所有行。 本质上是在每次提交事务后自动执行一个TRUNCATE命令。

`DROP`

在当前事务块的结尾将删除临时表。

`TABLESPACE` `_tablespace_name_`

指定新表将要在 `_tablespace_name_` 表空间内创建。如果没有声明，将咨询default_tablespace， 或如果该表为临时表，那么将使用temp_tablespaces。

`_query_`

一个SELECT、TABLE 或VALUES命令，或者一个运行预备好的 `SELECT` 、 `TABLE` 或 `VALUES` 查询的EXECUTE命令。

`WITH [ NO ] DATA`

这个子句指定查询产生的数据是否应该拷贝到新表中。如果不，那么只拷贝表结构。 缺省是拷贝数据。

注意

这条命令在功能上等效于SELECT INTO，但是更建议你用这个命令， 因为它不太可能和 `SELECT INTO` 语法的其它方面混淆。另外， `CREATE TABLE AS` 提供了 `SELECT INTO` 功能的超集。

在PostgreSQL 8.0 之前， `CREATE TABLE AS` 总是在它创建的表中包含 OID ， 而在 PostgreSQL 8.0 里， `CREATE TABLE AS` 命令允许明确声明是否应该包含 OID 。如果没有明确声明是否应该包含 OID ， 那么使用配置变量default_with_oids的设置。 到了PostgreSQL这个变量缺省为假， 缺省行为和 8.0 之前的版本不同。因此，那些要求 `CREATE TABLE AS` 创建的表包含 OID 的应用应该明确声明 `WITH (OIDS)` 以确保要求的行为。

例子

创建一个只包含表 `films` 中最近的记录的新表 `films_recent` ：

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

要完整的拷贝一个表，也可以使用 `TABLE` 命令的简易形式：

```
CREATE TABLE films2 AS
  TABLE films;
```

使用预备语句创建一个只包含表 `films` 中最近的记录的新临时表 `films_recent` ， 该临时表包含 OID 并且在事务结束时将被删除：

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

兼容性

`CREATE TABLE AS` 兼容SQL标准， 下面是非标准的扩展：

- 标准要求在子查询子句周围有圆括弧，在PostgreSQL 里，这些圆括弧是可选的。
- 在标准中， `WITH [ NO ] DATA` 子句是必选的；在PostgreSQL中，它是可选的。
- PostgreSQL处理临时表的方法和标准相差较大；参阅CREATE TABLE获取细节。
- `WITH` 子句是PostgreSQL扩展， 并且 SQL 标准中也没有存储参数和 OID 。
- PostgreSQL表空间的概念也不是标准的一部分。 因此 `TABLESPACE` 子句也是扩展。

又见

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

Name

CREATE TABLESPACE -- 定义一个新的表空间

Synopsis

```
CREATE TABLESPACE _tablespace_name_ [ OWNER _user_name_ ] LOCATION '_directory_'
```

描述

`CREATE TABLESPACE` 注册一个集群范围内的新表空间。 表空间的名字必须在该数据库集群中的任何现有表空间中唯一。

表空间允许超级用户在文件系统中定义一个可选的位置， 这个位置可以存放代表数据库对象的数据文件(比如表和索引)。

一个用户，如果有合适的权限，就可以把 `CREATE DATABASE` ，`CREATE TABLE` ，`CREATE INDEX` ，`ADD CONSTRAINT` 之一传递给 `_tablespace_name_` ， 这样就让这些对象的数据文件存储在指定的表空间里。

参数

`_tablespace_name_`

要创建的表空间的名字。这个名字不能以 `pg_` 开头， 因为这些名字是保留给系统表空间使用的。

`_user_name_`

将拥有这个表空间的用户名。如果省略，缺省为执行此命令的用户名。 只有超级用户可以创建表空间，但是他们可以把表空间的所有者授予非超级用户。

`_directory_`

用于表空间的目录。目录必须是空的，并且由运行PostgreSQL 系统用户所有。目录必须用一个绝对路径声明。

注意

只有在那些支持符号连接的系统上才支持表空间。

`CREATE TABLESPACE` 不允许在一个事务块内部执行。

例子

在 `/data/dbs` 创建一个表空间 `dbspace` ：

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

在 `/data/indexes` 创建一个表空间 `indexspace` 并由用户 `genevieve` 所有：

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

兼容性

`CREATE TABLESPACE` 是PostgreSQL扩展。

又见

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

Name

CREATE TEXT SEARCH CONFIGURATION -- 定义一个新的文本搜索配置

Synopsis

```
CREATE TEXT SEARCH CONFIGURATION _name_ (
    PARSER = _parser_name_ |
    COPY = _source_config_
)
```

描述

`CREATE TEXT SEARCH CONFIGURATION` 创建新的文本搜索配置。 一个文本搜索配置声明一个能将一个字符串划分成符号的文本搜索解析器， 加上可以用于确定搜索对哪些标记感兴趣的字典。

若仅声明分析器，那么新的文本搜索配置初始没有从符号类型到词典的映射， 因此会忽略所有的单词。后来的 `ALTER TEXT SEARCH CONFIGURATION` 命令必须用于创建映射来使得配置是有效的。或者，可以拷贝现有的文本搜索配置。

若模式名称已给出，那么文本搜索配置会在声明的模式中创建。否则会在当前模式创建。

定义文本搜索配置的用户成为其所有者。

查阅Chapter 12获取更多信息。

参数

`_name_`

要创建的文本搜索配置的名称。该名称可以有模式修饰。

`_parser_name_`

用于该配置的文本搜索分析器的名称。

`_source_config_`

要复制的现有文本搜索配置的名称。

注意

`PARSER`和`COPY`选项是互相排斥的, 因为当一个现有配置被复制,其分析器配置也被复制了。

兼容性

在SQL标准中没有`CREATE TEXT SEARCH CONFIGURATION`语句。

又见

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

Name

CREATE TEXT SEARCH DICTIONARY -- 定义一个新的文本搜索字典

Synopsis

```
CREATE TEXT SEARCH DICTIONARY _name_ (
    TEMPLATE = _template_
    [, _option_ = _value_ [, ... ]]
)
```

描述

`CREATE TEXT SEARCH DICTIONARY` 创建一个新的文本搜索字典。 一个文本搜索字典指定了一个发现查询中感兴趣或者不感兴趣的词的方式。 一个基于文本搜索模板的字典，它声明了实际上执行工作的函数。 通常字典提供了一些选项，这些选项控制模板函数的详细性能。

若给出模式名称，那么文本搜索字典就会在指定模式中创建。否则会在当前模式中创建。

创建文本搜索字典的用户将成为其所有者。

参阅Chapter 12获取更多详细信息。

参数

`_name_`

要创建的文本搜素字典的名称。名称可以有模式修饰。

`_template_`

定义字典基本行为的文本搜索模板的名称。

`_option_`

为该字典设置的特定模板选项的名称。

`_value_`

用于特定模板选项的值。如果该值不是一个简单的标识符或号码，它必须被引用 （如果您希望，您就可以一直引用它）。

这个选项可以以任何顺序出现。

例子

下面的示例命令用停用词的非标准列表创建了一个基于snowball的字典。

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

兼容性

在SQL标准中没有 `CREATE TEXT SEARCH DICTIONARY` 语句。

又见

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

Name

CREATE TEXT SEARCH PARSER -- 定义一个新的文本搜索的解析器

Synopsis

```
CREATE TEXT SEARCH PARSER _name_ (
    START = _start_function_ ,
    GETTOKEN = _gettoken_function_ ,
    END = _end_function_ ,
    LEXTYPES = _lextypes_function_
    [, HEADLINE = _headline_function_ ]
)
```

描述

`CREATE TEXT SEARCH PARSER` 定义一个新的文本搜索的解析器。 一个文本搜索解析器定义一个将文本字符串分解为符号的方法， 并且为符号设定类型（类）。一个解析器本身不是特别有用， 但必须连同一些文本搜索字典一起绑定到一个文本搜索配置中，用于搜索。

若给出模式名称，那么文本搜索解析器就会在指定模式中创建。否则会在当前模式中创建。

您必须是超级用户才能使用 `CREATE TEXT SEARCH PARSER` 。 （做这个限制的原因是一个错误的文本搜索解析器的定义会混淆甚至崩溃服务器。）

参阅Chapter 12获取更多信息。

参数

`_name_`

要创建的文本搜索解析器的名称。名称可以有模式修饰。

`_start_function_`

解析器启动函数的名称。

`_gettoken_function_`

为解析器获取下一个符号的函数的名称。

`_end_function_`

解析器结束函数的名称。

`_lextypes_function_`

解析器lextypes函数的名称（一个返回关于其产生的符号类型集的信息的函数）。

`_headline_function_`

解析器的标题函数的名称（一个总结一组符号的函数）。

如果必要的话，函数名称可以有模式修饰。没有给出参数类型， 因为函数的每种类型的参数列表是预定的。除了标题函数所有的都是必需的。

参数可以以任何顺序出现，不仅是上面所示的。

兼容性

在SQL中没有`CREATE TEXT SEARCH PARSER`语句。

又见

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

Name

CREATE TEXT SEARCH TEMPLATE -- 定义一个新的文本搜索模板

Synopsis

```
CREATE TEXT SEARCH TEMPLATE _name_ (
    [ INIT = _init_function_ , ]
    LEXIZE = _lexize_function_
)
```

描述

`CREATE TEXT SEARCH TEMPLATE`定义一个新的文本搜索模板。 文本搜索模板定义执行文本搜索字典的函数。一个模板本身无效的， 必须作为一个字典实例化才能使用。字典通常声明给定的参数到模板函数。

若给出模式名称，那么文本搜索模板会在声明的模式中创建。否则会在当前模式创建。

您必须是超级用户使用`CREATE TEXT SEARCH TEMPLATE`。 做这个限制的原因是一个错误的文本搜索模板定义可能会混淆甚至崩溃服务器。 从字典中分离模板的原因是模板封装定义字典的"不安全"方面。 在定义字典时可以设置的参数对于要设置的非特权用户来说是安全的， 因此创建一个字典不需要特权操作。

参阅Chapter 12获取更多信息。

参数

`_name_`

要创建的文本搜索模板的名称。名称可以有模式修饰。

`_init_function_`

模板初始函数的名称。

`_lexize_function_`

模板的lexize函数的名称。

若需要，函数名可以有模式修饰。未给出参数类型， 因为每个函数类型的参数列表都是预定的。lexize函数是必须的，而init函数是可选的。

参数可以以任何顺序出现，不仅仅是上面显示的那样。

兼容性

在SQL标准中没有`CREATE TEXT SEARCH TEMPLATE`语句。

又见

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRIGGER

Name

CREATE TRIGGER -- 定义一个新触发器

Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER _name_ { BEFORE | AFTER | INSTEAD OF } { _event_ [ OR ... ]
    ON _table_name_
    [ FROM _referenced_table_name_ ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( _condition_ ) ]
    EXECUTE PROCEDURE _function_name_ ( _arguments_ )

where `_event_` can be one of:

    INSERT
    UPDATE [ OF _column_name_ [, ... ] ]
    DELETE
    TRUNCATE
```

描述

`CREATE TRIGGER` 创建一个新的触发器。 触发器将与指定表或视图关联并且将在特定事件发生时执行声明的 `_function_name_` 函数。

触发器可以声明为在对记录进行操作之前(在检查约束之前和 `INSERT` 、 `UPDATE` 、 `DELETE` 执行前)；或操作完成之后(在检查约束之后和完成 `INSERT` 、 `UPDATE` 、 `UPDATE` 操作)；或取代操作 （在视图上插入、更新或删除）触发。如果触发器在事件之前或者取代事件， 触发器可能略过当前记录的操作或改变被插入的记录(只对 `UPDATE` 和 `UPDATE` 操作有效)。如果触发器在事件之后， 所有更改，包括其他触发器的影响，对触发器都是"可见"的。

一个被标记为 `FOR EACH ROW` 的触发器为操作修改的每一行都调用一次。 比如，一个影响 10 行的 `DELETE` 将导致任何在目标关系上的 `ON DELETE` 触发器独立调用 10 次，每个被删除的行调用一次。 相比之下，一个被标记为 `FOR EACH STATEMENT` 的触发器只执行一次， 而不管有多少行被修改。（特别是，一个修改零行的操作仍然会导致合适的 `FOR EACH STATEMENT` 触发器被执行。

指定为触发 `INSTEAD OF` 触发器事件的触发器必须被标记为 `FOR EACH ROW` ，并且只能在视图上定义。视图上的 `BEFORE` 和 `AFTER` 触发器必须被标记为 `FOR EACH STATEMENT` 。

另外，触发器可能被定义为为 `TRUNCATE` 触发， 尽管只有 `FOR EACH STATEMENT` 。

下面表总结中的触发器类型可能被用在表和视图上：

何时	事件	行级别	语句级别
`BEFORE`	`INSERT` / `UPDATE` / `DELETE`	表	表和视图
TRUNCATE	—	表	
`AFTER`	`INSERT` / `UPDATE` / `DELETE`	表	表和视图
TRUNCATE	—	表	
`INSTEAD OF`	`INSERT` / `UPDATE` / `DELETE`	视图	—
TRUNCATE	—	—	

还有，触发器定义可以声明一个布尔 `WHEN` 条件，用来测试触发器是否应该被触发。 在行级别触发器中， `WHEN` 条件可以检测该行的字段的旧的和/或新值。 语句级别的触发器也可以拥有 `WHEN` 条件，尽管该特性对于它们来说不太有用， 因为该条件不能引用表中的任何值。

如果多个同类型的触发器为同一事件做了定义，那么它们将按照字母顺序被触发。

当声明了 `CONSTRAINT` 选项时，这个命令创建一个约束触发器。 作为正规触发器也是相同的，除了触发器触发的时间可以使用SET CONSTRAINTS 调整。约束触发器必须是 `AFTER ROW` 触发器。 它们可以在导致触发事件的语句的结束触发，也可以在包含的事务的结束触发； 在后面一种情况下，它们被称为延迟的。 一个等待延迟的触发器触发也可以通过使用 `SET CONSTRAINTS` 强制立即发生。 当它们实现的约束非法时，约束触发器预计会引发一个异常。

`SELECT` 并不更改任何行，因此你不能创建 `SELECT` 触发器。这种场合下规则和视图更合适些。

请参考Chapter 36获取更多触发器信息。

参数

`_name_`

赋予新触发器的名称。它必需和任何作用于同一表的触发器不同。 该名字不能是模式修饰的—触发器继承它的表的模式。 对于约束触发器，当使用 `SET CONSTRAINTS` 修改触发器的行为时， 这也是要使用的名字。

`BEFORE``AFTER` `INSTEAD OF`

决定该函数是在事件之前、之后还是取代事件时调用。 约束触发器只能被声明为 `AFTER` 。

`_event_`

`INSERT` 、 `UPDATE` 、 `DELETE` 或 `TRUNCATE` 之一。它声明激发触发器的事件。 多个事件可以用 `OR` 声明。

对于 `UPDATE` 事件，使用这个语法声明一个字段列表是可能的：

```
UPDATE OF _column_name1_ [, _column_name2_ ... ]
```

该触发器将只在至少一个列表中的字段在 `UPDATE` 命令的目标中提及时触发。

`INSTEAD OF UPDATE` 事件不支持字段的列表。

`_table_name_`

触发器作用的表或视图的名称(可以有模式修饰)

`_referenced_table_name_`

约束引用的另外一个表的名字（可以有模式修饰）。这个选项用于外键约束， 不推荐用于一般用途。只能为约束触发器指定。

`DEFERRABLE` `NOT DEFERRABLE` `INITIALLY IMMEDIATE` `INITIALLY DEFERRED`

触发器的默认时机。参阅CREATE TABLE文档获取这些约束选项的详细信息。 只能为约束触发器指定。

`FOR EACH ROW` `FOR EACH STATEMENT`

这些选项声明触发器过程是否为触发器事件影响的每个行触发一次， 还是只为每条 SQL 语句触发一次。如果都没有声明， 那么 `FOR EACH STATEMENT` 将是缺省。 约束触发器只能声明为 `FOR EACH ROW` 。

`_condition_`

一个决定触发器函数实际上是否执行的布尔表达式。如果声明了 `WHEN` ， 那么该函数只有 `_condition_` 返回 `true` 时被调用。在 `FOR EACH ROW` 触发器中， `WHEN` 条件可以通过分别写 `OLD.``_column_name_` 或 `NEW.``_column_name_` 参考字段的旧的和/或新的行值。 当然， `INSERT` 触发器不能参考 `OLD` ， `OLD` 触发器不能参考 `NEW` 。

`INSTEAD OF` 触发器不支持 `WHEN` 条件。

目前， `WHEN` 表达式不能包含子查询。

请注意，对于约束触发器， `WHEN` 条件的计算是不延迟的， 只是在行更新操作执行之后立即发生。如果该条件计算不为真， 那么触发器就不排队延迟执行。

`_function_name_`

一个用户提供的函数，它声明为不接受参数并且返回 `trigger` 类型， 该函数将在触发器被触发时调用。

`_arguments_`

一个可选的用逗号分隔的参数列表，它将在触发器执行的时候提供给函数。 这些参数是文本字符串常量。也可以在这里写简单的名字和数值常量， 但是它们会被转换成字符串。请检查该触发器函数的实现语言的描述， 找出如何在该函数中访问这些参数；这些参数可能和普通的函数参数不同。

注意

要在表上创建一个触发器，用户必需在该表上有 `TRIGGER` 权限。 用户也必须在触发器函数上有 `EXECUTE` 权限。

使用DROP TRIGGER删除触发器。

字段特有的触发器（使用 `UPDATE OF` `_column_name_` 定义的）将在它的任意字段作为目标列出在 `UPDATE` 命令的 `SET` 列表中时触发。当触发器没有触发时，字段的值也是有可能改变的，因为通过 `BEFORE UPDATE` 触发器做的行内容的改变是不考虑的。相反的， 命令如 `UPDATE ... SET x = x ...` 将触发在字段 `x` 上的触发器，尽管字段的值没有改变。

在 `BEFORE` 触发器中， `WHEN` 条件只在函数被或将被执行之前计算， 所以使用 `WHEN` 与在触发器函数的开始测试相同的条件并无实质区别。 要特别的注意，条件看到的 `NEW` 行是当前的值，可能被早些的触发器修改了。 另外， `BEFORE` 触发器的 `WHEN` 条件不允许检测 `NEW` 行的系统字段(比如 `oid`)，因为那些目前还没有设置。

在 `AFTER` 触发器中， `WHEN` 条件只在行更新发生之后计算， 并且它决定一个事件是否在语句的最后排队触发该触发器。所以当 `AFTER` 触发器的 `WHEN` 条件没有返回真时，不需要排队一个事件， 也不需要在语句的最后重新抓取行。如果触发器只需要为少量的行触发， 这会导致修改许多行的语句明显的加速。

在PostgreSQL 7.3以前， 必须把触发器函数声明为返回 `opaque` 占位类型，而不是 `trigger` 类型。 为了支持加载老的转储文件， `CREATE TRIGGER` 将接受一个声明为返回 `opaque` 的函数，但是它将发出一条 NOTICE 并且把函数声明的返回类型改成 `trigger` 。

例子

当表 `accounts` 的一行要被更新时，执行函数 `check_account_update` ：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

同样的，但是只在字段 `balance` 在 `UPDATE` 命令的目标中指定时执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

这种形式只在字段 `balance` 实际上改变了值时执行该函数：

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

只在改变了什么东西时，调用函数记录 `accounts` 的更新：

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

为每一行执行函数 `view_insert_row` 以在一个视图下插入行到表：

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

Section 36.4包含一个完整的用C写的触发器函数的例子。

兼容性

PostgreSQL里的 `CREATE TRIGGER` 语句实现了一个SQL标准的子集。目前仍然缺少下面的功能：

- SQL 允许你为"old"和"new"行或者表定义别名， 用于定义触发器的动作(也就是 `CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername`)。 因为PostgreSQL 允许触发器过程以任意数量的用户定义语言进行书写， 所以访问数据的工作是用和语言相关的方法实现的。
- PostgreSQL不允许旧的和新的表在语句级别的触发器中引用， 也就是，包含所有旧的和/或新的行的表，在SQL标准中被 `OLD TABLE` 和 `NEW TABLE` 子句提及。
- PostgreSQL只允许为触发的动作执行用户定义的函数。 SQL 标准允许执行一些其它的命令，比如拿 `CREATE TABLE` 作为触发器动作。这个限止并不难绕开，只要创建一个执行这些命令的用户定义的函数即可。

SQL 要求多个触发器应该以创建的时间顺序执行。 PostgreSQL采用的是按照名字顺序，并认为这样更加方便。

SQL 要求必须在级联 `DELETE` 完成之后再触发级联删除上的 `BEFORE DELETE` 触发器。PostgreSQL 的行为是 `BEFORE DELETE` 将永远在删除动作之前触发， 即使对于级联删除也是如此，我们认为这样更一致。如果 `BEFORE` 触发器在由引用操作引起的更新期间修改行或阻止更新，仍然存在不标准的行为。 这将导致违反约束或者存储不符合参照完整性的数据。

用 `OR` 给一个触发器声明多个动作是 PostgreSQL对SQL标准的扩展。

触发触发器 `TRUNCATE` 的能力是一个 PostgreSQL对SQL标准的扩展，就像在视图上定义语句级别触发器的能力。

`CREATE CONSTRAINT TRIGGER` 是一个 PostgreSQL对SQL标准的扩展。

又见

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

Name

CREATE TYPE -- 定义一个新数据类型

Synopsis

```
CREATE TYPE _name_ AS
    ( [ _attribute_name_ _data_type_ [ COLLATE _collation_ ] [, ... ] ] )

CREATE TYPE _name_ AS ENUM
    ( [ '_label_' [, ... ] ] )

CREATE TYPE _name_ AS RANGE (
    SUBTYPE = _subtype_
    [ , SUBTYPE_OPCLASS = _subtype_operator_class_ ]
    [ , COLLATION = _collation_ ]
    [ , CANONICAL = _canonical_function_ ]
    [ , SUBTYPE_DIFF = _subtype_diff_function_ ]
)

CREATE TYPE _name_ (
    INPUT = _input_function_,
    OUTPUT = _output_function_
    [ , RECEIVE = _receive_function_ ]
    [ , SEND = _send_function_ ]
    [ , TYPMOD_IN = _type_modifier_input_function_ ]
    [ , TYPMOD_OUT = _type_modifier_output_function_ ]
    [ , ANALYZE = _analyze_function_ ]
    [ , INTERNALLENGTH = { _internallength_ | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = _alignment_ ]
    [ , STORAGE = _storage_ ]
    [ , LIKE = _like_type_ ]
    [ , CATEGORY = _category_ ]
    [ , PREFERRED = _preferred_ ]
    [ , DEFAULT = _default_ ]
    [ , ELEMENT = _element_ ]
    [ , DELIMITER = _delimiter_ ]
    [ , COLLATABLE = _collatable_ ]
)

CREATE TYPE _name_
```

描述

`CREATE TYPE` 为当前数据库注册一个新的数据类型。 定义该类型的用户成为其所有者。

如果给出模式名，那么该类型是在指定模式中创建。否则它将在当前模式中创建。 类型名必需和同一模式中任何现有的类型或者域不同。 （因为表和数据类型有联系， 所以类型名也不能和同模式中的表名字冲突。）

有5种形式的 `CREATE TYPE`， 显示在上面的语法摘要里。 他们分别创建复合类型、枚举类型、范围类型、基本类型或壳类型。 前四个在下面依次讨论。壳类型是简单的一个稍后定义的类型的占位符；通过发出没有参数只有类型名的 `CREATE TYPE` 创建。 当创建范围类型和基本类型时，壳类型作为向前引用需要，在下面章节中讨论。

复合类型

`CREATE TYPE` 的第一种形式创建一个复合类型。 复合类型是通过一列属性名和数据类型声明的。如果它的数据类型可排序， 那么也可以指定属性的排序。复合类型本质上和一个表的行类型一样， 但是如果只是想定义一个类型，那么使用 `CREATE TYPE` 就可以避免直接创建实际的表。一个独立的复合类型是有用的，例如， 做为一个函数的参数或者返回类型。

要想能够创建一个复合类型，必须在所有的属性类型上有 `USAGE` 权限。

枚举类型

`CREATE TYPE` 的第二种形式创建一个枚举（enum）类型， 在Section 8.7中描述。枚举类型接受一个或更多的引用标签的列表， 每个标签必须小于 `NAMEDATALEN` 字节长度 （在标准的PostgreSQL建立中是64字节）。

范围类型

`CREATE TYPE` 的第三种形式创建一个新的范围类型， 在Section 8.17中描述。

范围类型的 `_subtype_` 可以是有一个相关的b-tree操作符类（决定范围类型值的顺序）的任意类型。 通常子类型的缺省b-tree操作符类用于决定顺序；要使用一个非缺省的操作符类，用 `_subtype_opclass_` 指定它的名字。 如果子类型是可排序的，并且你希望在范围的排序中使用非缺省的排序， 那么使用 `_collation_` 选项指定想要的排序。

可选的 `_canonical_` 函数必须接受一个被定义的范围类型的参数，并且返回相同类型的值。适用时，这用于转换范围类型到标准形式。参阅Section 8.17.8 获取更多信息。创建一个 `_canonical_` 函数比较棘手，因为它必须在范围类型可以被声明之前定义。要做到这点，必须先创建一个壳类型，壳类型是一个除了名字和所有者之外没有其他属性的占位符类型。这可以通过发出没有其他额外参数的 `CREATE TYPE` `_name_` 命令来完成。 然后可以使用该壳类型作为参数和结果声明该函数，最后可以使用相同的名字声明范围类型。 这将自动使用有效的范围类型替代壳类型条目。

可选的 `_subtype_diff_` 函数必须接受两个 `_subtype_` 类型的值作为参数， 并且返回 `双精度` 值表示两个给定值之间的不同。 虽然这是可选的，但是提供它允许GiST索引在范围类型字段上有更大的效率。 参阅Section 8.17.8获取更多信息。

基本类型

`CREATE TYPE` 的第四种形式创建一个新的基本类型(标量类型)。 要创建一个新的基本类型，你必须是一个超级用户。 （做这个限制是因为一个错误的类型定义会混淆或者甚至崩溃服务器。）

参数可以按任意顺序出现，而不是上面显示的那样，并且大多数都是可选的。 必须在定义类型之前先用 `CREATE FUNCTION` 注册两个或更多个函数。 支持函数 `_input_function_` 和 `_output_function_` 是必须的， 而函数 `_receive_function_` 、 `_send_function_` 、 `_type_modifier_input_function_` 、 `_type_modifier_output_function_` 和 `_analyze_function_` 是可选的。 通常，这些函数必须用 C 或者其它低层语言编写。

`_input_function_` 函数将该类型的外部文本形式转换成可以被该类型的操作符和函数识别的内部形式。 `_output_function_` 用途相反。 输入函数可以声明为接受一个类型为 `cstring` 的参数， 或者接受三个类型分别为 `cstring` 、 `oid` 、 `integer` 的参数。第一个参数是 C 字符串形式的输入文本， 第二个参数是该类型自身的 OID(数组类型除外，这种情况下它们接受自身元素的类型 OID)， 第三个是目标字段的 `typmod` (如果未知则传递 -1)。 输入函数必须返回一个自身数据类型的值。通常，输入函数应当被声明为 STRICT ， 否则当读取 NULL 输入时将被使用第一个参数为 NULL 进行调用， 并且必须仍然返回 NULL 或报错。 （这个特性主要是为了支持域输入函数， 这种函数可能需要拒绝 NULL 输入。）输出函数必须被声明为接受一个新数据类型的参数， 并且必须返回 `cstring` 类型。输出函数不会被使用 NULL 调用。

可选的 `_receive_function_` 把该类型的外部二进制表现形式转换成内部表现形式。如果没有提供这个函数， 那么该类型不能用二进制输入。二进制格式应该选取那种比较容易转换同时还有一定移植性的内部格式。比如，标准的整数数据类型使用网络字节序作为 外部的二进制表现形式，而内部表现形式则是机器的本机字节序。例如， 接收函数应该声明为接受一个类型为 `internal` 的参数， 或者是三个类型分别为 `internal` 、 `oid` 、 `integer` 的参数。第一个参数是一个指向一个 `StringInfo` 缓冲区的、保存接受字节串的指针；可选的参数和文本输入函数一样。 接收函数必须返回一个该类型的数据值。通常接收函数应当被声明为 STRICT ，否则否则当读取 NULL 输入时将被使用第一个参数为 NULL 进行调用， 并且必须仍然返回 NULL 或报错。这个特性主要是为了支持域接收函数， 这种函数可能需要拒绝 NULL 输入。同样，可选的 `_send_function_` 把类型的内部表现形式转换为外部二进制表现形式。如果没有提供这些函数， 那么类型就不能用二进制方式输出。 发送函数必须声明为接收一个新数据类型并且必须返回 `bytea` 结果。 发送函数不会被以 NULL 值调用。

这个时候你应该觉得奇怪，输入和输出函数怎么可以声明为返回新类型的结果或者是接受新类型的参数， 而且是在新类型创建之前就需要创建它们。答案是类型必须被首先定义为一个壳类型， 它只是一个除了名称和属主之外没有其他属性的占位符类型。这可以通过没有其他额外参数的 `CREATE TYPE` `_name_` 命令来完成。 然后就可以引用该壳类型定义输入输出函数。最后， `CREATE TYPE` 把这个壳类型替换成完整的、有效的类型定义，这样就可以使用新类型了。

如果支持类型修饰符，那么可选的 `_type_modifier_input_function_` 和 `_type_modifier_output_function_` 是需要的，这是附属于类型声明的可选的约束， 如 `char(5)` 和 `numeric(30,2)` 。PostgreSQL 允许用户定义的类型接受一个或多个简单的约束

或标识符作为修饰符。不过，这个信息必须能够装进一个非负的整型值里，以在系统表中存储。 `_type_modifier_input_function_` 以 `cstring` 数组的形式传送声明的修饰符。它必须检查值的有效性（如果值是错误的则抛出一个错误），如果是正确的，则返回一个非负的 `integer` 值，该值将被作为字段"typmod"存储。如果类型没有 `_type_modifier_input_function_`， 那么类型修饰符将被拒绝。 `_type_modifier_output_function_` 转换内部的整数typmod值为用户显示的正确形式。它必须返回一个 `cstring` 值，该值是附加到类型名之后的准确的字符串；如 `numeric` 的函数返回 `(30,2)` 。 允许省略 `_type_modifier_output_function_`， 省略的情况下，缺省显示形式是只有存储的typmod整型值包含在圆括号中。

可选的 `_analyze_function_` 为该数据类型的字段执行与该类型相关的统计信息收集。缺省时， 如果该类型有个缺省的 B-tree 操作符类，那么 `ANALYZE` 将尝试使用该类型的"等于"和"小于"操作符收集信息。 对于非标量类型，这种行为很可能不合适，因此可以通过提供一个自定义的分析函数覆盖它。 分析函数必须声明为接收单独一个 `internal` 类型的参数， 并且返回一个 `boolean` 结果。分析函数的详细 API 在 `src/include/commands/vacuum.h` 里。

尽管新类型的内部表现形式只有输入输出函数和其它你创建来使用该类型的函数了解， 但内部表现形式还是有几个属性必须为PostgreSQL声明。 `_internallength_` 是最重要的一个。基本数据类型可定义成为定长，这时 `_internallength_` 是一个正整数，也可以是变长的，通过把 `_internallength_` 设为 `VARIABLE` 表示。(在内部，这个状态是通过将 `typlen` 设置为-1实现的。)所有变长类型的内部形式都必须以一个四字节整数开头， 这个整数给出此类型这个数值的全长。

可选的标记 `PASSEDBYVALUE` 表明该类型的数值是按值而不是引用传递。 你不能传递那些内部形式大于 `Datum` 类型尺寸(大多数机器上是 4 字节， 有些是 8 字节)的数据类型的值。

`_alignment_` 参数声明该数据类型要求的对齐存储方式。 允许的数值等效于按照 1, 2, 4, 8 字节边界对齐。请注意变长类型必须有至少 4 字节的对齐， 因为它们必须包含一个 `int4` 作为第一个部分。

`_storage_` 参数允许为变长数据类型选择存储策略(定长类型只允许使用 `plain`)。 `plain` 声明该数据类型总是用内联的方式而不是压缩的方式存储。 `extended` 声明系统将首先试图压缩一个长的数据值， 然后如果它仍然太长的话就将它的值移出主表，但系统将不会压缩它。 `external` 声明禁止系统进行压缩并且允许将它的值移出主表。 `main` 允许压缩，但是不赞成把数值移动出主表(如果实在不能放在一行里的话， 仍将移动出主表，它比 `extended` 和 `external` 项更愿意保存在主表里)。

`_like_type_` 参数为指定数据类型的基本代表属性提供一个可选的方式：从某些现有的类型中拷贝他们。 `_internallength_` 、 `_passedbyvalue_` 、 `_alignment_` 和 `_storage_` 的值从命名的类型中拷贝。 （这是可能的，尽管通常不需要，通过和 `LIKE` 子句一起指定他们， 来覆盖一些这些值。）这种方式指定代表，在新类型"piggybacks" 的低级实现以某种方式在现有类型上时尤其有用。

`_category_` 和 `_preferred_` 参数可以用于帮助控制哪个隐式转换将被用于模糊的情况。 每个数据类型都属于一个由单个ASCII字符命名的类，并且每个类型不是"preferred" 就是不在它的类中。当这个规则对于解析重载函数或操作符有帮助时， 解析器更愿意转换为优先的类型（但是只限于来自不在相同类中的其他类型）。 更多信息请参阅Chapter 10。对于没有隐式转换到或来自任意其他类型的类型， 保留这些设置作为缺省就足够了。不过，对于有隐式转换的的相关类型的组， 通常标记他们都属于一个类并且在该类中选取一个或两个"最一般" 的类型作为优先是有帮助的。当添加一个用户定义的类型到一个现有的内建类时， `_category_` 参数尤其有用， 如数值型或字符串类型。但是，也有可能创建新的完全用户定义的类型类别。 选取任意ASCII字符而不是一个大写的字母来命名这样一个类别。

如果用户希望字段的数据类型缺省时不是 NULL ，那么可以在 `DEFAULT` 关键字里声明一个缺省值(可以被附着在特定字段上的 `DEFAULT` 子句覆盖)。

用 `ELEMENT` 关键字声明数组元素的类型。 比如， `ELEMENT = int4` 定义了一个 4 字节整数(`int4`)的数组。有关数组类型的更多细节在下面描述。

可用 `_delimiter_` 指定用于这种类型数组的外部形式的数值之间的分隔符。 缺省的分隔符是逗号(`,`)。 请注意分隔符是和数组元素类型相关联，而不是数组类型本身。

如果可选的布尔参数 `_collatable_` 为真， 那么该类型的字段定义和表达式可以通过使用 `COLLATE` 子句携带排序信息。取决于在该类型上函数操作符的实现，以实际上利用排序信息； 仅仅通过标记该类型可排序，这就不会自动发生。

数组类型

在创建用户定义类型的时候，PostgreSQL 自动创建一个与之关联的数组类型，其名字由该元素类型的名字前缀一个下划线组成， 并且如果有必要保持它小于 `NAMEDATALEN` 字节长度时截断。 （如果这样生成的名字与一个现有的类型名冲突，那么重复该进程， 知道找到一个不冲突的名字。）这个隐含创建的数组类型是变长并且使用内建的 `array_in` 和 `array_out` 输入和输出函数。 数组类型追踪它的元素类型的所有者或模式的任意更改， 并且如果元素类型有更改时删除。

你很可能会问如果系统自动制作正确的数组类型，那为什么还要有个 `ELEMENT` 选项? 使用 `ELEMENT` 的场合有一：你定义的定长类型碰巧在内部是一个一定数目相同事物的数组， 而你又想允许这 N 个事物可以通过下标直接访问，除了某些操作符将把该类型当做整体进行处理。 比如，类型 `point` 表示为两个浮点数，每个可以用 `point[0]` 和 `point[1]` 访问。请注意这个功能只适用于定长类型， 并且其内部形式是一个相同定长域的序列。一个可以下标化的变长类型必须有被 `array_in` 和 `array_out` 使用的一般化的内部表现形式。 出于历史原因（也就是，这是明显错误的，但是要改变它却太晚了）， 定长数组类型的下标从 0 开始，而不是像变长数组那样的从 1 开始。

参数

`_name_`

将要创建的类型名(可以有模式修饰)

`_attribute_name_`

复合类型的一个属性(字段)的名字

`_data_type_`

要成为一个复合类型的字段的现有数据类型的名字

`_collation_`

与复合类型或范围类型字段有关的现有排序的名字

`_label_`

表示与枚举类型的值有关的文本标签的字符串字面值

`_subtype_`

元素类型的名字，范围类型将代表的范围

`_subtype_operator_class_`

该子类型的b-tree操作符类的名字

`_canonical_function_`

范围类型的标准化函数的名字

`_subtype_diff_function_`

子类型的差异函数的名字

`_input_function_`

一个函数的名称，将数据从外部文本形式转换成内部格式。

`_output_function_`

一个函数的名称，将数据从内部格式转换成适于显示的外部文本形式。

`_receive_function_`

一个函数的名称，把数据从类型的外部二进制形式转换成内部形式

`_send_function_`

一个函数的名称，把数据从类型的内部形式转换成外部二进制形式

`_type_modifier_input_function_`

一个函数的名称，把修饰符的数组类型转换成内部形式

`_type_modifier_output_function_`

一个函数的名称，把类型的修饰符的内部形式转换成外部形式

`_analyze_function_`

为该数据类型执行统计分析的函数名

`_internallength_`

一个数值常量，说明新类型的内部表现形式的字节长度。缺省假定它是变长的。

`_alignment_`

该数据类型的存储对齐要求。如果声明了，必须是 `char` 、 `int2` 、 `int4` (缺省)、 `double` 之一。

`_storage_`

该数据类型的存储策略。如果声明了，必须是 `plain` (缺省)、 `external` 、 `extended` 、 `main` 之一。

`_like_type_`

与新类型将要有的表现相同的现有数据类型的名字。 `_internallength_` 、 `_passedbyvalue_` 、 `_alignment_` 、 `_storage_` 的值是从该类型中拷贝的， 除非在 `CREATE TYPE` 命令中明确的说明覆写。

`_category_`

这个类型的类别代码（单个ASCII字符）。"用户定义类型"却省是 `'U'` 。 其他标准类别代码可以在Table 47-52中找到。 你也可以选择其他ASCII字符来创建自定义类别。

`_preferred_`

如果这个类型在它的类型类别中是首选类型则为真，否则为假。缺省是假。 在一个现有的类型类别中创建一个新的首选类型时要非常小心， 因为这会导致行为上意外的变化。

`_default_`

该类型的缺省值。若省略则为 NULL

`_element_`

被创建的类型是数组；这个声明数组元素的类型。

`_delimiter_`

数组元素之间分隔符

`_collatable_`

如果这个类型的操作可以使用排序信息则为真。缺省为假。

注意

因为一旦类型被创建之后对它的使用就没有限制， 所以创建一个基本类型或范围类型就等价于授予所有用户执行类型定义中指定的各个函数的权限， 这对于大多数类型定义中指定的函数来说不会造成什么不良问题。 但是如果你设计的新类型在内部形式和外部形式之间转换的时候使用"敏感信息"， 那么你仍然要再三考虑、多加小心。

PostgreSQL版本8.3之前， 生成的数组类型的名字总是正好是元素类型的名字前置一个下划线字符(`_`)。 （因此限制类型名字的长度比其他名字的字符要少。） 虽然这仍然是通常的情况， 但是数组类型名字可能会有变化，假使最大长度名字或与下划线开始的用户类型名字冲突。 依赖于这个约定的书写代码因此弃用了。取而代之，使用 `pg_type` . `typarray` 来定位与一个给定类型相关的数组类型。

避免使用以下划线开始的类型和表名是明智的。虽然服务器将改变生成的数组类型名， 以避免与用户给定的名字冲突，但是仍然有混淆的更显， 尤其是老的客户端软件可能假设以下划线开始的类型名总是代表数组。

PostgreSQL版本8.2之前，壳类型创建语法 `CREATE TYPE` `_name_` 并不存在。 创建新的基本类型之前必须首先创建其输入函数。这样，PostgreSQL 将会首先把新类型的名字看作输入函数的返回类型并隐含创建壳类型， 然后这个壳类型将被随后定义的输入输出函数引用。这种老式的方法目前仍然被支持， 但已经反对使用，将来可能不再支持。同样，为了避免函数定义中的临时壳类型偶然地搞乱系统表， 当输入函数用 C 语言书写时，将只能用这种方法创建壳类型。

在PostgreSQL 7.3 以前，要通过使用占位伪类型 `opaque` 代替函数的前向引用来避免创建壳类型。7.3 之前 `cstring` 参数和结果同样需要声明为 `opaque` 。要支持加载旧的转储文件， `CREATE TYPE` 将接受那些用 `opaque` 声明的输入输出函数， 但是它将发出一条通知并且用正确的类型改变函数的声明。

例子

这个例子创建一个复合类型并且在一个函数定义中使用它：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

这个命令创建枚举类型，并且将它用于一个表定义：

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

这个例子创建一个范围类型：

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

这个命令创建 `box` 基本数据类型，并且将这种类型用于一个表定义：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果 `box` 的内部结构是一个四个 `float4` 的数组，可以使用：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

来允许一个 box 的数值成分成员可以用下标访问。否则该类型和前面的行为一样。

这条命令创建一个大对象类型并将其用于一个表定义：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

更多的例子，包括合适的输入和输出函数，位于Section 35.11。

兼容性

`CREATE TYPE` 命令的第一种形式，创建一个复合类型，符合SQL标准。其他形式是PostgreSQL 的扩展。SQL标准中的 `CREATE TYPE` 语句也定义了没有在PostgreSQL中实现的其他形式。

用0属性创建一个复合类型的能力是PostgreSQL 与标准具体的偏差（类似于 `CREATE TABLE` 中的相同情况）。

又见

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

Name

CREATE USER -- 定义一个新数据库角色

Synopsis

```
CREATE USER _name_ [ [ WITH ] _option_ [ ... ] ]

 这里的`_option_`可以是:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT _connlimit_
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD '_password_'
    | VALID UNTIL '_timestamp_'
    | IN ROLE _role_name_ [, ...]
    | IN GROUP _role_name_ [, ...]
    | ROLE _role_name_ [, ...]
    | ADMIN _role_name_ [, ...]
    | USER _role_name_ [, ...]
    | SYSID _uid_
```

描述

`CREATE USER` 现在是CREATE ROLE的别名。 唯一的区别是 `CREATE USER` 命令缺省假设有 `LOGIN` ， 而 `CREATE ROLE` 缺省是 `NOLOGIN` 。

兼容性

`CREATE USER` 语句是PostgreSQL扩展。 SQL 标准把用户的定义交给了实现来完成。

又见

CREATE ROLE

CREATE USER MAPPING

Name

CREATE USER MAPPING -- 定义一个新的用户到外部服务器的映射

Synopsis

```
CREATE USER MAPPING FOR { _user_name_ | USER | CURRENT_USER | PUBLIC }
    SERVER _server_name_
    [ OPTIONS ( _option_ '_value_' [ , ... ] ) ]
```

描述

`CREATE USER MAPPING` 定义了一个用户到外部服务器的映射。 一个用户映射通常封装连接信息， 外部数据封装器与外部服务器封装的信息一起使用来访问外部数据资源。

外部服务器的所有者可以为任意用户创建服务器的用户映射。另外， 如果服务器上的 `USAGE` 权限已经授予一个用户， 那么该用户可以为其自身用户名创建一个用户映射，。

参数

`_user_name_`

映射到外部服务器的现有用户的名称。 `CURRENT_USER` 和 `USER` 匹配当前用户的名称。当指定 `PUBLIC` 时，一个所谓的公共映射就创建了， 当没有特定用户的映射适用时就会使用该映射。

`_server_name_`

一个现有服务器的名称，用户映射就是为其创建的。

`OPTIONS (` `_option_` ' `_value_` ' [, ...])

该子句声明用户映射的选项。该选项通常定义映射的实际用户名和密码。 选项名称必须是唯一的。允许的选项名和值特定于服务器的外部数据封装器。

例子

为用户 `bob`，服务器 `foo` 创建一个用户映射:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

兼容性

`CREATE USER MAPPING` 遵循ISO/IEC 9075-9 (SQL/MED)。

又见

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

Name

CREATE VIEW -- 定义一个新视图

Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW _name_ [ ( _column_name_ [,
    [ WITH ( _view_option_name_ [= _view_option_value_] [, ... ] ) ]
    AS _query_
```

描述

`CREATE VIEW` 定义一个查询的视图。 这个视图不是物理上实际存在的，并且在该视图每次被引用的时候都会运行一次查询。

`CREATE OR REPLACE VIEW` 类似，不过如果一个同名的视图已经存在， 那么将替换它。新查询必须生成与现有视图查询生成的字段相同的字段（也就是， 相同的字段名字，相同的顺序和相同的数据类型），但是可能添加额外的字段到列表的结尾。 该计算导致输出字段可能完全不同。

如果给出了一个模式名(比如 `CREATE VIEW myschema.myview ...`)， 那么该视图将在指定的模式中创建，否则将在当前模式中创建。 临时视图存在于一个特殊的模式里，所以创建临时视图的时候，不能给出模式名。 新视图名字必需和同一模式中任何其它视图、表、序列、索引或外部表的名字不同。

参数

`TEMPORARY` 或 `TEMP`

如果声明了这个子句，那么视图就以临时视图的方式创建。 临时视图在当前会话结束的时候将被自动删除。当前会话中，在临时视图存在的期间， 将无法看到现有的同名关系，除非用模式修饰的名字引用它们。

如果视图引用的任何基础表是临时的，那么视图将被创建为临时的 (不管是否声明了 `TEMPORARY`)。

`RECURSIVE`

创建一个递归的视图。语法

```
CREATE RECURSIVE VIEW _name_ (_columns_) AS SELECT _..._;
```

等同于

```
CREATE VIEW _name_ AS WITH RECURSIVE _name_ (_columns_) AS (SELECT _..._) SELECT _columns
```

必须为一个递归的视图指定一个视图字段列表。

`_name_`

所要创建的视图名称(可以有模式修饰)。

`_column_name_`

一个可选的名字列表，用作视图的字段名。如果没有给出，字段名取自查询。

`WITH (` `_view_option_name_` [= `_view_option_value_`] [, ...])

该子句为视图指定选项参数；目前，唯一支持的参数名是 `security_barrier` ， 当视图打算提供行级安全时， 应该启用该参数。参阅Section 38.5 获取全部细节。

`_query_`

一个将为视图提供行和列的SELECT 或VALUES语句。

注意

使用DROP VIEW语句删除视图。

请注意视图字段的名字和类型不一定是你们期望的那样。比如,

```
CREATE VIEW vista AS SELECT 'Hello World';
```

在两个方面很糟糕：字段名缺省是 `?column?` 并且字段的数据类型缺省是 `unknown` 。 如果你想视图的结果是一个字符串文本，那么请像下面这样使用：

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

对视图引用的表的访问的权限由视图的所有者决定。在一些情况下， 这可用于提供安全但是限制访问底层表。不过，不是所有视图对于篡改都是安全的；参阅Section 38.5获取细节。在视图里调用的函数当作他们直接从使用视图的查询里调用看待。因此， 视图的用户必须有调用视图使用的所有函数的权限。

当 `CREATE OR REPLACE VIEW` 用在一个现有的视图上时， 只改变了视图定义的SELECT规则。其他视图属性，包括所有权、权限和非SELECT规则， 保持不变。要替换视图，你必须拥有该视图（包括成为拥有者角色的一员）。

可更新的视图

简单的视图是自动可更新的：系统允许 `INSERT` 、 `UPDATE` 和 `DELETE` 语句和在规则表上一样的方式在视图上使用。 如果视图满足所有下列的条件，那么就是自动可更新的：

- 视图在它的 `FROM` 列表中必须只有一个条目， 该条目必须是一个表或其他可更新视图。
- 视图定义必须没有在顶级包含 `WITH` 、 `DISTINCT` 、 `GROUP BY` 、 `HAVING` 、 `LIMIT` 或 `OFFSET` 子句。
- 视图定义必须没有在顶级包含集合运算（ `UNION` 、 `INTERSECT` 或 `EXCEPT` ）。
- 视图的选择列表中的所有字段必须简单的引用底层关系的字段。 它们不能是表达式、字面值或函数。也不能引用系统字段。
- 在视图的选择列表中，底层关系的字段出现不能超过一次。
- 视图必须没有 `security_barrier` 属性。

如果视图是自动可更新的，那么系统将转换视图上的任意 `INSERT` 、 `UPDATE` 或 `DELETE` 语句为相应的底层基本关系上的语句。

如果一个自动可更新的视图包含一个 `WHERE` 条件， 那么该条件约束基本关系的哪些行可以被视图上的 `UPDATE` 和 `DELETE` 语句修改。不过，允许 `UPDATE` 更改一个行，所以它不再满足 `WHERE` 条件，并且因此不再通过视图可见。 相似的， `INSERT` 命令可以潜在的插入不满足 `WHERE` 条件的基本关系行，并且因此不能通过视图可见。

不满足所有这些条件的更复杂的视图缺省是只读的：系统将不允许在该视图上插入、 更新或删除。可以通过在该视图上创建 `INSTEAD OF` 触发器获得可更新视图的效果， 该触发器必须转换在该视图上的尝试插入等为其他表上的适当动作。更多信息请参见 CREATE TRIGGER。还有一种可能性是创建规则 （参阅CREATE RULE），但是实际上触发器更容易理解和正确使用。

请注意，用户在视图上执行插入、更新或删除必须在该视图上有相应的插入、 更新或删除的权限。此外，视图的所有者必须在底层基础关系上有相关的权限， 但是执行更新的用户不需要在底层基础关系上的任何权限 （参阅Section 38.5）。

例子

创建一个由所有喜剧电影组成的视图：

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

这将创建一个视图，在视图创建的时候包含 `film` 表中字段。 尽管用 `*` 创建了该视图，但是后来添加到表中的字段将不会是视图的一部分。

创建一个由数字1到100组成的递归的视图：

```
CREATE RECURSIVE VIEW nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

兼容性

SQL 标准为 `CREATE VIEW` 声明了一些附加的功能：

```
CREATE VIEW _name_ [ ( _column_name_ [, ...] ) ]
    AS _query_
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

完整的 SQL 命令可选的子句是：

`CHECK OPTION`

这个选项控制自动可更新视图的行为。给出时，对视图的 `INSERT` 和 `UPDATE` 都要检查以确保新行满足视图定义的条件（也就是说， 新行应该可以通过视图看到）。如果没有通过检查，更新将被拒绝。 没有 `CHECK OPTION` ，允许对视图的 `INSERT` 和 `UPDATE` 命令创建通过该视图不可见的行。 （后者的行为当前只有PostgreSQL提供。）

`LOCAL`

对这个视图进行完整性检查。

`CASCADED`

对此视图和任何相关视图进行完整性检查。在既没有声明 `CASCADED` 也没有声明 `LOCAL` 时，假设为 `CASCADED` 。

`CREATE OR REPLACE VIEW` 是PostgreSQL 的扩展。临时视图的概念也是扩展。 `WITH` 子句也是一个扩展。

又见

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

Name

DEALLOCATE -- 删除一个预备语句

Synopsis

```
DEALLOCATE [ PREPARE ] { _name_ | ALL }
```

描述

`DEALLOCATE` 用于删除前面编写的预备语句。 如果你没有明确删除一个预备语句，那么它将在会话结束的时候被删除。

有关预备语句的更多信息。参阅PREPARE。

参数

`PREPARE`

这个关键字总被忽略。

`_name_`

将要删除的预备语句。

`ALL`

删除所有预备语句。

兼容性

SQL 包括一个 `DEALLOCATE` 语句，但它只是用于嵌入式 SQL 。

参见

EXECUTE, PREPARE

DECLARE

Name

DECLARE -- 定义一个游标

Synopsis

```
DECLARE _name_ [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR _query_
```

描述

`DECLARE` 允许用户创建游标，用于在一个大的查询里面检索少数几行数据。 在游标建立后，使用FETCH可以从中取出行。

> **Note:** 本页描述游标在SQL命令级别的使用。如果你试图在一个PL/pgSQL函数里面使用游标， 结果往往是不同的；参阅Section 40.7。

参数

`_name_`

将要创建的游标名。

`BINARY`

令游标以二进制而不是文本格式返回数据。

`INSENSITIVE`

表明从游标检索出来的数据不应该被发生在游标创建后的游标的更新动作影响。 在PostgreSQL里，这是缺省行为；这个关键字没有什么作用， 只在为了和 SQL 标准兼容的时候接受它。

`SCROLL``NO SCROLL`

`SCROLL` 声明该游标可以用于以倒序的方式检索数据行(也就是反向检索)。 根据查询的执行计划的不同，声明 `SCROLL` 可能会对查询的执行时间有不良影响。 `NO SCROLL` 声明该游标不能用于以倒序的方式检索数据行。 缺省仅允许在某些情况下倒序检索，这不同于指定 `SCROLL` 。参见注意获取细节。

`WITH HOLD` `WITHOUT HOLD`

`WITH HOLD` (缺省)声明该游标可以在创建它的事务成功提交后继续使用。 `WITHOUT HOLD` 声明该游标不能在创建它的事务之外使用。

`_query_`

一个SELECT或VALUES命令，它提供游标返回的行。

`BINARY` , `INSENSITIVE` , `SCROLL` 关键字可以以任何顺序出现。

注意

通常游标和 `SELECT` 一样返回文本格式。 `BINARY` 选项声明游标应该返回二进制格式的数据。这样就减少了服务器和客户端的转化工作， 花费程序员更多的工作解决平台依赖的二进制数据格式问题。 比如，如果查询从某个整数列返回 1 ，在缺省的游标里将获得一个字符串 `1` ，但在二进制游标里将得到一个 4 字节的包含该数值内部形式的数值(大端顺序)。

应该小心使用二进制游标。一些客户端应用(比如psql) 是不能识别二进制游标的，它们期望返回的数据是文本格式。

> **Note:** 如果客户端应用使用"扩展查询"协议发出 `FETCH` 命令， 那么 Bind 协议声明数据是用文本还是用二进制格式检索。这个选择覆盖游标的定义。 因此，在使用扩展查询协议的时候，二进制游标的概念已经过时了， 任何游标都可以当作文本或者二进制的格式发出。

如果没有声明 `WITH HOLD` ，那么这个命令创建的游标只能在当前事务中使用。 这样，不带 `WITH HOLD` 的 `DECLARE` 在事务块外面没有任何用处 ： 游标将一直存活到事务结束。因此，PostgreSQL 将在这样的命令出现在事务块外面的时候报错。使用BEGIN, COMMIT和ROLLBACK定义一个事务块。

如果声明了 `WITH HOLD` 并且创建该游标的事务成功提交， 那么游标还可以在同一会话随后的事务里访问。但如果创建它的事务回滚， 那么游标被删除。带 `WITH HOLD` 创建的游标是用一个明确的 `CLOSE` 命令或者是会话终止来关闭的。在目前的实现里， 由一个游标代表的行是被拷贝到一个临时文件或者内存区里的， 这样他们就仍然可以在随后的事务中被访问。

当查询包含 `FOR UPDATE` 或 `FOR SHARE` 的时候， 不能指定 `WITH HOLD` 。

在定义一个要用来反向抓取的游标的时候，应该声明 `SCROLL` 选项， 这是 SQL 标准要求的。 不过，为了和早期的版本兼容， 只要游标的查询计划简单得不需要额外的开销，PostgreSQL 在没有声明 `SCROLL` 的时候也允许反向抓取。不过， 建议应用开发人员不要依赖于使用没有使用 `SCROLL` 定义的游标的反向查找功能。如果声明了 `NO SCROLL` ， 那么不管怎样都会禁止反向抓取的功能。

当查询包含 `FOR UPDATE` 或 `FOR SHARE` 时也不允许反向抓取；因此在这种情况下不能声明 `SCROLL` 。

> **Caution**
>
> 如果调用任何不稳定的函数，那么可以回滚并且 `WITH HOLD` 游标可能给出意外的结果（参阅xref linkend="xfunc-volatility">）。 当重新抓取到以前抓取到的行时，函数可能会重新执行，可能导致结果与之前的不同。 一个绕开这种情况的方法是声明 `WITH HOLD` 游标， 并且在从它读取任何行之前提交事务。这将强制游标的整个输出在临时存储中物化， 这样不稳定的函数精确的每行只执行一次。

如果游标的查询包括 `FOR UPDATE` 或 `FOR SHARE` ， 那么返回的行在它们第一次被抓取的时候锁定，和有这个选项的SELECT 一样。另外，返回行将是最新的版本；因此提供这些选项相当于SQL标准调用一个 "敏感的游标"。（和 `FOR UPDATE` 或 `FOR SHARE` 一起指定 `INSENSITIVE` 是错误的。）

> **Caution**
>
> 如果游标定义为使用 `UPDATE ... WHERE CURRENT OF` 或 `DELETE ... WHERE CURRENT OF` ，通常推荐使用 `FOR UPDATE` 。 使用 `FOR UPDATE` 阻止其他会话在抓取和更新之间改变行。没有 `FOR UPDATE` ， 如果在创建游标后改变了行，那么随后的 `WHERE CURRENT OF` 命令将没有作用。

使用 `FOR UPDATE` 的另外一个原因是，没有它，如果游标查询不符合SQL标准的 "简单可更新"原则，那么随后的 `WHERE CURRENT OF` 可能会失败 （尤其是，游标必须只引用一个表，并且不使用分组或 `ORDER BY` ）。 不是简单可更新的游标可能会也可能不会工作，取决于计划选择细节；所以在最坏的情况下， 应用可能在测试中工作而在生产中失败。

不和 `WHERE CURRENT OF` 一起使用 `FOR UPDATE` 的主要原因是， 你需要游标是可回滚的，或对随后的更新不敏感的（也就是说，持续显示旧的数据）。 如果需要这样，请注意上面显示的注意事项。 |

SQL 标准中的游标只能在嵌入SQL(ESQL)的应用中使用。 PostgreSQL服务器没有一个明确的 `OPEN` 语句；一个游标被认为在定义时就已经打开了。不过，PostgreSQL 嵌入的 SQL 预处理器(ECPG)支持 SQL 标准的习惯， 包括那些和 `DECLARE` 和 `OPEN` 相关的语句。

可以通过查询 `pg_cursors` 系统视图看到所有可用游标。

例子

定义一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

参阅FETCH获取有关游标使用的更多例子。

兼容性

SQL标准说它是依赖于实现的，不管游标对缺省的底层数据的当前更新是否敏感。 在PostgreSQL中，游标缺省是敏感的，并且可以通过声明 `FOR UPDATE` 使其敏感。其他产品工作可能不同。

SQL 标准只允许在嵌入的SQL中和模块中使用游标。 PostgreSQL允许交互地使用游标。

二进制游标是PostgreSQL扩展。

又见

CLOSE, FETCH, MOVE

DELETE

Name

DELETE -- 删除一个表中的行

Synopsis

```
[ WITH [ RECURSIVE ] _with_query_ [, ...] ]
DELETE FROM [ ONLY ] _table_name_ [ * ] [ [ AS ] _alias_ ]
    [ USING _using_list_ ]
    [ WHERE _condition_ | WHERE CURRENT OF _cursor_name_ ]
    [ RETURNING * | _output_expression_ [ [ AS ] _output_name_ ] [, ...] ]
```

描述

`DELETE` 从指定的表里删除满足 `WHERE` 子句的行。 如果 `WHERE` 子句不存在，将删除表中所有行。结果是一个有效的空表。

> **Tip:** TRUNCATE是一个PostgreSQL扩展， 它提供一个更快的从表中删除所有行的机制。

使用数据库中其它表的信息删除某个表中的数据行有两个办法：使用子查询， 或者在 `USING` 子句中声明额外的表。哪种技巧更合适取决于特定的环境。

可选的 `RETURNING` 子句将使得 `DELETE` 计算并返回实际被删除了的行。 任何使用表字段的表达式和/或 `USING` 中提到的其他表的字段， 都可以用于计算。 `RETURNING` 列表的语法和 `SELECT` 输出列表的语法相同。

要对表进行删除，你必须对它有 `DELETE` 权限，同样也必须有 `USING` 子句的表以及 `_condition_` 上读取的表的 `SELECT` 权限。

参数

`_with_query_`

`WITH` 子句允许指定一个或多个可以通过 `DELETE` 查询中的名字引用的子查询。 参阅Section 7.8和SELECT获取详细信息。

`_table_name_`

要删除行的表的名字（可以有模式修饰）。如果在表的名字前指定了 `ONLY`， 则只从指定的表中删除匹配的行。如果没有指定 `ONLY`，则从指定的表及其所有子表中的删除匹配的行。可选的，可以在表名的后面指定 `*` 以明确的指出包括后代表。

`_alias_`

目标表的别名。如果提供了别名，那么它将完全掩盖实际的表名。例如给定 `DELETE FROM foo AS f` 之后， `DELETE` 语句的剩余部分必须使用 `f` 而不是 `foo` 来引用该表。

`_using_list_`

表表达式列表，允许来自其他表的列出现在 `WHERE` 条件中。这与可以在 `SELECT` 命令的 *FROM 子句* 中指定的表列表相似。例如，可以为该表的名字声明一个别名。不要在 `_using_list_` 里重复目标表，除非你希望产生一个自连接。

`_condition_`

一个返回 `boolean` 值的表达式，只用表达式返回 `true` 的行被删除。

`_cursor_name_`

在 `WHERE CURRENT OF` 条件中使用的游标的名字。要删除的行是最近从这个游标获取到的。 该游标必须是一个在 `DELETE` 的目标表中非分组的查询。请注意， `WHERE CURRENT OF` 不能和一个布尔条件一起指定，参阅DECLARE获取更多关于和 `WHERE CURRENT OF` 一起使用游标的信息。

`_output_expression_`

计算并在删除行后由 `DELETE` 命令返回的一个表达式。该表达式可以使用由 `_table_name_` 命名的表的任意字段名或在 `USING` 中列出的表。 `*` 返回所有字段。

`_output_name_`

用于返回的列名称。

输出

成功时， `DELETE` 命令返回形如

```
DELETE _count_
```

的标签。 `_count_` 是被删除的行数。 请注意，当删除被 `BEFORE DELETE` 触发器取消时，这个数字可能小于匹配 `_condition_` 的行数。 如果 `_count_` 为 0 则没有行被该查询删除， 这个不认为是错误。

如果 `DELETE` 命令包含一个 `RETURNING` 子句，那么其结果非常类似于 `SELECT` 语句基于 `RETURNING` 子句中包含的字段和值列表的结果， 只是基于被删除的行进行计算而已。

注意

PostgreSQL允许你在 `WHERE` 条件里引用其它表的字段， 方法是在 `USING` 子句里声明其它表。比如，要删除给出制片商制作的所有电影，可以：

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

这里实际发生的事情是在 `films` 和 `producers` 之间的一个连接， 然后所有成功连接的 `films` 行都标记为删除。这个语法不是标准的，更标准的语法是这么做：

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

有时候连接风格比子查询风格更容易写或者执行更快。

例子

删除所有电影(films)但不删除音乐(musicals)：

```
DELETE FROM films WHERE kind <> 'Musical';
```

清空 `films` 表：

```
DELETE FROM films;
```

从 tasks 表及其子表中删除，并返回所有被删除的行：

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

删除游标 `c_tasks` 当前指向的 `tasks` 的行：

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

兼容性

这条命令遵循SQL标准，但是 `USING` 和 `RETURNING` 子句是PostgreSQL的扩展，就像在 `DELETE` 中使用 `WITH` 。

DISCARD

Name

DISCARD -- 丢弃会话状态

Synopsis

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

描述

`DISCARD`释放与数据库会话相关的内部资源。这些资源通常在会话结束时释放。

`DISCARD TEMP` 删除所有在当前会话中创建的临时表。 `DISCARD PLANS` 释放所有的内部缓存查询计划。 `DISCARD ALL` 重置一个会话到初始状态,丢弃临时资源和新设置的本地会话的改变。

参数

`TEMPORARY` or `TEMP`

删除在当前会话中创建的所有的临时表。

`PLANS`

释放所有缓存的查询计划。

`ALL`

释放所有与当前会话相关的临时资源并重置到其初始状态。 当前，这与执行以下语句序列有相同的效果:

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
```

说明

`DISCARD ALL` 在一个事务内部模块中不能被执行。

兼容性

`DISCARD` 是一个PostgreSQL扩展。

DO

Name

DO -- 执行匿名代码块

Synopsis

```
DO [ LANGUAGE _lang_name_ ] _code_
```

描述

`DO` 执行一段匿名代码块, 换句话说，在程序语言过程中一次性执行的匿名函数。

代码块被看做是没有参数的一段函数体，返回值类型是 `void` 。它的解析和执行时同一时刻发生的。

可选属性 `LANGUAGE` 可以在代码块之前写，也可以写在代码块的后面。

参数

`_code_`

程序语言代码可以被执行的。程序语言必须指定为字符串才行, 就像命令 `CREATE FUNCTION` ，推荐使用美元符号一样。

`_lang_name_`

用来解析代码的程序语言的名字，如果缺省，默认的语言是 `plpgsql` 。

注意事项

程序语言在使用之前，必须通过命令 `CREATE LANGUAGE` 安装到当前的数据库中。 `plpgsql` 是默认的安装语言，其它语言安装时必须指定。

如果语言是不受信任的，用户必须有使用程序语言的 `USAGE` 权限，或者是超级用户。 在语言上，这同创建一个函数是一样的权限要求。

例子

授予角色 `webuser` 对模式 `public` 下视图的所有操作权限：

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.ta
    END LOOP;
END$$;
```

兼容性

SQL标准中没有 `DO` 语句。

又见

CREATE LANGUAGE

DROP AGGREGATE

Name

DROP AGGREGATE -- 删除一个聚集函数

Synopsis

```
DROP AGGREGATE [ IF EXISTS ] _name_ ( _argtype_ [ , ... ] ) [ CASCADE | RESTRICT ]
```

描述

`DROP AGGREGATE` 删除一个现存的聚集函数。执行这条命令的用户必须是该聚集函数的所有者。

参数

`IF EXISTS`

如果指定的聚集不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

现存的聚集函数名(可以有模式修饰)

`_argtype_`

聚集函数操作的输入数据类型，要引用一个零参数聚集函数，请用 `*` 代替输入数据类型列表。

`CASCADE`

级联删除依赖于这个聚集函数的对象

`RESTRICT`

如果有任何依赖对象，则拒绝删除这个聚集函数。这是缺省处理。

例子

将 `integer` 类型的聚集函数 `myavg` 删除：

```
DROP AGGREGATE myavg(integer);
```

兼容性

SQL 标准里没有 `DROP AGGREGATE` 语句。

又见

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

Name

DROP CAST -- 删除一个类型转换

Synopsis

```
DROP CAST [ IF EXISTS ] (_source_type_ AS _target_type_) [ CASCADE | RESTRICT ]
```

描述

`DROP CAST` 删除一个先前定义过的类型转换。

要能删除一个类型转换，你必须拥有源或者目的数据类型。这是和创建一个类型转换相同的权限。

参数

`IF EXISTS`

如果指定的转换不存在，那么发出一个 notice 而不是抛出一个错误。

`_source_type_`

类型转换里的源数据类型。

`_target_type_`

类型转换里的目标数据类型。

`CASCADE``RESTRICT`

这些键字没有任何效果，因为在类型转换上没有依赖关系。

例子

删除从 `text` 到 `int` 的转换：

```
DROP CAST (text AS int);
```

兼容性

`DROP CAST` 遵循 SQL 标准。

又见

CREATE CAST

DROP COLLATION

Name

DROP COLLATION -- 删除一个排序规则

Synopsis

```
DROP COLLATION [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP COLLATION` 删除一个已经定义的排序规则。你必须是该排序规则的所有者才能删除它。

参数

`IF EXISTS`

如果指定的排序规则不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

排序规则名（可以有模式修饰)。

`CASCADE`

自动删除依赖于这个排序规则的对象。

`RESTRICT`

如果有任何依赖对象，则拒绝删除这个排序规则。这是缺省处理。

例子

要删除名为 `german` 的排序规则：

```
DROP COLLATION german;
```

兼容性

`DROP COLLATION` 遵循SQL标准， 但是 `IF EXISTS` 选项是PostgreSQL的扩展。

又见

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

Name

DROP CONVERSION -- 删除一个编码转换

Synopsis

```
DROP CONVERSION [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP CONVERSION` 删除一个先前定义过的编码转换。要想删除一个转换， 你必须拥有该转换。

参数

`IF EXISTS`

如果指定的转换不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

编码转换的名字(可以用模式修饰)。

`CASCADE``RESTRICT`

这些关键字没有作用，因为编码转换上没有依赖关系。

例子

删除一个叫做 `myname` 的编码转换：

```
DROP CONVERSION myname;
```

兼容性

SQL 标准里没有 `DROP CONVERSION` 语句，但是有一个与 `CREATE TRANSLATION` 同在的 `DROP TRANSLATION` 语句， `CREATE TRANSLATION` 语句类似于PostgreSQL中的 `CREATE CONVERSION` 语句。

又见

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

Name

DROP DATABASE -- 删除一个数据库

Synopsis

```
DROP DATABASE [ IF EXISTS ] _name_
```

描述

`DROP DATABASE` 删除一个数据库。删除一个现存数据库的目录入口并且删除包含数据的目录。只有数据库所有者能够执行这条命令。还有，如果你或者任何其他人正在与目标数据库连接， 那么就不能执行这条命令。所以要与 `postgres` 或者任何其它数据库连接，再发出这条命令。

`DROP DATABASE` 不能撤销，小心使用！

参数

`IF EXISTS`

如果指定的数据库不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要被删除的现有数据库名。

注意

`DROP DATABASE` 不能在事务块中执行。

这条命令在和目标数据库连接时不能执行。通常更好的做法是用dropdb 程序代替，该程序是此命令的一个封装。

兼容性

SQL 标准里没有 `DROP DATABASE` 语句。

又见

CREATE DATABASE

DROP DOMAIN

Name

DROP DOMAIN -- 删除一个域

Synopsis

```
DROP DOMAIN [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP DOMAIN` 删除一个域。只有域的所有者才能删除它。

参数

`IF EXISTS`

如果指定的域不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

一个现有的域(可以有模式修饰)。

`CASCADE`

级联删除依赖该域的对象（例如表字段）。

`RESTRICT`

如果有任何依赖对象存在，则拒绝删除此域。这个是缺省。

例子

删除 `box` 域：

```
DROP DOMAIN box;
```

兼容性

这条命令兼容 SQL 标准，但 `IF EXISTS` 选项是一个PostgreSQL扩展。

又见

CREATE DOMAIN, ALTER DOMAIN

DROP EXTENSION

Name

DROP EXTENSION -- 删除一个扩展

Synopsis

```
DROP EXTENSION [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP EXTENSION` 命令从数据库中删除一个扩展。 在删除扩展的过程中，构成扩展的组件也会一起删除。

必须是扩展的拥有者才能够使用 `DROP EXTENSION` 命令。

参数

`IF EXISTS`

当使用 `IF EXISTS` 参数，如果扩展不存在时，不会抛出错误，而是产生一个通知。

`_name_`

已经安装的扩展模块的名称。

`CASCADE`

自动删除依赖于该扩展的对象。

`RESTRICT`

如果有依赖于扩展的对象，则不允许删除次扩展（除非它所有的成员对象和其它扩展对象在一条 `DROP` 命令一起删除）。 这是缺省行为。

例子

从当前数据库中删除扩展 `hstore`

```
DROP EXTENSION hstore;
```

在当前数据库中，如果有使用 `hstore` 的对象的，这条命令就会失败，比如 任一表中的字段使用 `hstore` 类型。这时增加 `CASCADE` 选项会强制删除扩展和 依赖于扩展的对象。

兼容性

`DROP EXTENSION` 是PostgreSQL的扩展。

又见

CREATE EXTENSION, ALTER EXTENSION

DROP EVENT TRIGGER

Name

DROP EVENT TRIGGER -- 删除一个事件触发器

Synopsis

```
DROP EVENT TRIGGER [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP EVENT TRIGGER` 删除一个已存在的事件触发器。 当前用户是事件触发器的所有者才能够执行这个命令。

参数

`IF EXISTS`

当使用 `IF EXISTS`,如果事件触发器不存在时，不会抛出错误，而是产生一个通知。

`_name_`

待删除的事件触发器的名称。

`CASCADE`

自动删除依赖于事件触发器的对象。

`RESTRICT`

如果有依赖于事件触发器的对象，则不允许删除这个事件触发器。这是缺省行为。

例子

删除事件触发器 `snitch`：

```
DROP EVENT TRIGGER snitch;
```

兼容性

SQL标准中不支持 `DROP EVENT TRIGGER` 语句

又见

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP FOREIGN DATA WRAPPER

Name

DROP FOREIGN DATA WRAPPER -- 删除一个外部数据封装

Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP FOREIGN DATA WRAPPER` 删除一个已存在的外部数据封装器。 当前用户必须是外部数据封装器的所有者才能够执行此命令。

参数

`IF EXISTS`

当使用 `IF EXISTS`,如果外部数据封装器不存在时，不会抛出错误，而是产生一个通知。

`_name_`

已存在的外部数据封装器的名称。

`CASCADE`

自动删除依赖外部数据封装器的对象（如服务器）。

`RESTRICT`

如果有依赖于外部数据封装器的对象，则不允许删除外部数据封装器。这是缺省行为。

例子

删除外部数据封装器 `dbi`：

```
DROP FOREIGN DATA WRAPPER dbi;
```

兼容性

`DROP FOREIGN DATA WRAPPER`兼容ISO/IEC 9075-9 (SQL/MED)标准。 `IF EXISTS`选项是PostgreSQL的扩展。

又见

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

Name

DROP FOREIGN TABLE -- 删除一个外部表

Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP FOREIGN TABLE` 删除一个外部表。只有外部表的所有者才能够删除外部表。

参数

`IF EXISTS`

当使用 `IF EXISTS` ,如果外部表不存在时，不会抛出错误，而是产生一个通知。

`_name_`

待删除的外部表的名称（可以有模式修饰）

`CASCADE`

自动删除依赖于外部表的对象（如视图）

`RESTRICT`

如果有依赖于外部表的对象，则不允许删除外部表。这是缺省行为。

例子

删除外部表 `films` 和 `distributors` ：

```
DROP FOREIGN TABLE films, distributors;
```

兼容性

此命令兼容ISO/IEC 9075-9 (SQL/MED)标准。只不过标准只允许一条命令删除一个表。 `IF EXISTS` 选项也是PostgreSQL的扩展。

又见

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

Name

DROP FUNCTION -- 删除一个函数

Synopsis

```
DROP FUNCTION [ IF EXISTS ] _name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] )
    [ CASCADE | RESTRICT ]
```

描述

`DROP FUNCTION` 将删除一个现存的函数。要执行这条命令，用户必须是函数的所有者。 必须声明函数的参数类型，因为几个不同的函数可能会有同样的名字和不同的参数列表。

参数

`IF EXISTS`

如果指定的函数不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

现存的函数名称(可以有模式修饰)。

`_argmode_`

参数的模式： `IN` (缺省), `OUT` , `INOUT` , `VARIADIC` 。 请注意 `DROP FUNCTION` 实际上并不注意 `OUT` 参数， 因为判断函数的身份只需要输入参数。因此列出 `IN` , `INOUT` 和 `VARIADIC` 参数就足够了。

`_argname_`

参数的名字。请注意 `DROP FUNCTION` 实际上并不注意参数的名字， 因为判断函数的身份只需要输入参数的数据类型。

`_argtype_`

如果有的话，是函数参数的类型(可以用模式修饰)。

`CASCADE`

级联删除依赖于函数的对象(比如操作符或触发器)。

`RESTRICT`

如果有任何依赖对象存在，则拒绝删除该函数。这个是缺省。

例子

这条命令删除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

兼容性

SQL 标准里定义了一个 `DROP FUNCTION` 语句。但和这条命令不兼容。

又见

CREATE FUNCTION, ALTER FUNCTION

DROP GROUP

Name

DROP GROUP -- 删除一个数据库角色

Synopsis

```
DROP GROUP [ IF EXISTS ] _name_ [, ...]
```

描述

`DROP GROUP` 现在是DROP ROLE的别名。

兼容性

SQL 标准里没有 `DROP GROUP` 语句。

又见

DROP ROLE

DROP INDEX

Name

DROP INDEX -- 删除索引

Synopsis

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP INDEX` 从数据库中删除一个现存的索引。要执行这个命令，你必须是索引的所有者

参数

`CONCURRENTLY`

在表上删除索引的同时不对查询，插入，更新，删除加锁。 普通的 `DROP INDEX` 会使表获得独占锁，阻塞其他的访问，直到索引删除完成。使用此选项, 会由阻塞变为等待，直到其他冲突事物结束。

使用此选项时有几个注意事项。 只能指定一个索引的名称， 并且 `CASCADE` 选项不支持。 (同时, 一个索引是 `UNIQUE` or `PRIMARY KEY` 约束的时候，不能以此方式删除。) 同时，正规的 `DROP INDEX` 命令可以在事物内执行, 但是 `DROP INDEX CONCURRENTLY` 不可以在事物内执行.

`IF EXISTS`

如果指定的索引不存在，那么发出一个 notice 而不是抛出一个错误

`_name_`

要删除的索引名(可以有模式名字)。

`CASCADE`

级联删除依赖于该索引的对象。

`RESTRICT`

如果有依赖对象存在，则拒绝删除该索引。这个是缺省。

Examples

此命令将删除 `title_idx` 索引:

```
DROP INDEX title_idx;
```

兼容性

`DROP INDEX` 是 PostgreSQL 语言扩展。在 SQL 标准里没有索引的规定。

请参阅

CREATE INDEX

DROP LANGUAGE

Name

DROP LANGUAGE -- 删除一个过程语言

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP LANGUAGE` 删除曾注册过的过程语言name。 必须是超级用户或该过程语言的所有者才能使用 `DROP LANGUAGE` 。

> **Note:** 自PostgreSQL 9.1起，大多数过程语言成为了"extensions"， 因此删除时应该使用DROP EXTENSION，不能使用 `DROP LANGUAGE` 。

参数

`IF EXISTS`

如果指定的过程语言不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

现存语言的名称。出于向下兼容的考虑，这个名字可以用单引号包围。

`CASCADE`

级联删除依赖于该语言的对象(比如该语言写的函数)。

`RESTRICT`

如果存在依赖对象，则拒绝删除。这个是缺省。

例子

下面命令删除 `plsample` 语言：

```
DROP LANGUAGE plsample;
```

兼容性

SQL 标准里没有 `DROP LANGUAGE` 语句。

又见

ALTER LANGUAGE, CREATE LANGUAGE, droplang

DROP MATERIALIZED VIEW

Name

DROP MATERIALIZED VIEW -- 删除一个物化视图

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP MATERIALIZED VIEW` 删除一个现存的物化视图。要执行这个命令，你必须是物化视图的所有者。

参数

`IF EXISTS`

如果指定的物化视图不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的物化视图（可以有模式修饰）。

`CASCADE`

级联删除依赖于该物化视图的对象（例如其他物化视图或普通视图）。

`RESTRICT`

如果有依赖对象存在，则拒绝删除该物化视图。这个是缺省。

例子

此命令将删除 `order_summary` 物化视图：

```
DROP MATERIALIZED VIEW order_summary;
```

兼容性

`DROP MATERIALIZED VIEW` 是PostgreSQL的一个扩展。

又见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

Name

DROP OPERATOR -- 删除一个操作符

Synopsis

```
DROP OPERATOR [ IF EXISTS ] _name_ ( { _left_type_ | NONE } , { _right_type_ | NONE } ) [
```

描述

`DROP OPERATOR` 语句从数据库中删除一个现存的操作符。 要执行这个命令，你必须是操作符所有者。

参数

`IF EXISTS`

如果指定的操作符不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

一个现存的操作符的名字(可以有模式修饰)。

`_left_type_`

该操作符左操作数的类型。如果没有则写 `NONE` 。

`_right_type_`

该操作符右操作数的类型。如果没有则写 `NONE` 。

`CASCADE`

级联删除依赖于此操作符的所有对象。

`RESTRICT`

如果有任何依赖对象则拒绝删除此操作符。这个是缺省。

例子

将用于 `integer` 的幂操作符 `a^b` 删除：

```
DROP OPERATOR ^ (integer, integer);
```

为类型 `bit` 删除左单目位操作符 `~b` ：

```
DROP OPERATOR ~ (none, bit);
```

删除用于 `bigint` 的阶乘 `x!` ：

```
DROP OPERATOR ! (bigint, none);
```

兼容性

SQL 标准里没有 `DROP OPERATOR` 语句。

又见

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

Name

DROP OPERATOR CLASS -- 删除一个操作符类

Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] _name_ USING _index_method_ [ CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR CLASS` 删除一个现有操作符类。要执行这条命令， 你必须是此操作符类的所有者。

`DROP OPERATOR CLASS` 不删除被类引用的任何操作符或函数。 如果有任何索引依赖于该操作符类，必须声明 `CASCADE` 来全部删除。

参数

`IF EXISTS`

如果指定的操作符类不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

一个现存操作符类的名字(可以用模式修饰)。

`_index_method_`

操作符类所引用的索引访问方法的名字。

`CASCADE`

级联删除依赖于该操作符类的对象。

`RESTRICT`

如果有任何依赖对象存在，则拒绝删除此操作符类。这个行为是缺省。

注意

`DROP OPERATOR CLASS` 不会删除包含该类的操作符族，即使没有其他东西存在于族中也一样（尤其是，族是由 `CREATE OPERATOR CLASS` 创建的时）。一个空的操作符族是无害的， 但是为了整洁你可能希望用 `DROP OPERATOR FAMILY` 删除这个族； 或更直接的在一开始就使用 `DROP OPERATOR FAMILY` 。

例子

删除 B-tree 操作符类 `widget_ops` ：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果有任何现存的索引使用这个操作符类，那么这条命令将不能执行。 增加一个 `CASCADE` 删除这样的索引以及这个操作符类。

兼容性

SQL 标准里没有 `DROP OPERATOR CLASS` 语句。

又见

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

Name

DROP OPERATOR FAMILY -- 删除一个操作符族

Synopsis

```
DROP OPERATOR FAMILY [ IF EXISTS ] _name_ USING _index_method_ [ CASCADE | RESTRICT ]
```

描述

`DROP OPERATOR FAMILY` 删除一个现有的操作符族。 要执行这条命令，你必须是此操作符族的所有者。

`DROP OPERATOR FAMILY` 包含删除任何包含在该族内的操作符类， 但是不会删除被该族引用的任何操作符或函数。如果有任何索引依赖于该族内的操作符类， 则需要声明 `CASCADE` 来一起删除。

参数

`IF EXISTS`

如果指定的操作符族不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

一个现存操作符族的名字(可以用模式修饰)。

`_index_method_`

操作符族所引用的索引访问方法的名字

`CASCADE`

级联删除依赖于该操作符族的对象。

`RESTRICT`

如果有任何依赖对象存在，则拒绝删除此操作符族。这个行为是缺省。

例子

删除 B-tree 操作符类 `float_ops` ：

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果有任何现存的索引使用该族中的操作符类，那么这条命令将不能执行。 增加一个 `CASCADE` 删除这样的索引以及这个操作符族。

兼容性

SQL 标准里没有 `DROP OPERATOR FAMILY` 语句。

又见

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

Name

DROP OWNED -- 删除一个数据库角色所拥有的数据库对象

Synopsis

```
DROP OWNED BY _name_ [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP OWNED` 删除一个数据库角色所拥有的所有数据库对象。 所有该角色在当前数据库里和共享对象（数据库，表空间） 上的所有对象上的权限都将被撤销。

参数

`_name_`

将要删除所拥有对象并且撤销其权限的角色名。

`CASCADE`

级联删除所有依赖于被删除对象的对象。

`RESTRICT`

拒绝删除那些有任何依赖对象存在的对象。这个是缺省。

注意

`DROP OWNED` 常用来做删除角色前的准备工作。 因为 `DROP OWNED` 仅会对当前数据库产生影响， 所以通常需要在每个该角色拥有对象的数据库上执行一次。

使用 `CASCADE` 选项可能损害到其他用户所拥有的对象。

REASSIGN OWNED命令可以用来重新分配某个或某些角色所拥有对象的属主。

该角色拥有的数据库和表空间将被删除。

兼容性

`DROP OWNED` 语句是一个PostgreSQL扩展。

又见

REASSIGN OWNED, DROP ROLE

DROP ROLE

Name

DROP ROLE -- 删除一个数据库角色

Synopsis

```
DROP ROLE [ IF EXISTS ] _name_ [, ...]
```

描述

`DROP ROLE` 删除指定的角色。要删除一个超级用户角色， 你自己必须也是一个超级用户；要删除非超级用户角色，你必须有 `CREATEROLE` 权限。

不能删除仍然被集群中的任意数据库引用的角色，如果想删除它，会抛出一个错误。 在删除一个角色之前，你必须删除它拥有的所有对象(或者重新赋予他们新的所有者)， 并且撤销赋予该角色的任何权限。REASSIGN OWNED和 DROP OWNED命令可以达到这个目的。

不过，没有必要删除涉及该角色的角色成员关系； `DROP ROLE` 自动撤销目标角色在任何其它角色里面的成员关系，以及其它角色在目标角色里的成员关系。 其它角色不会被删除，也不会受到其它影响。

参数

`IF EXISTS`

如果指定的角色不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的角色名字。

注意

PostgreSQL包含了一个dropuser程序， 有着和这个命令相同的功能(实际上，它调用这个命令)，但是它可以从命令行上运行。

例子

删除一个角色：

```
DROP ROLE jonathan;
```

兼容性

SQL 标准定义了`DROP ROLE`，但它只允许每次删除一个角色， 并且它声明了和PostgreSQL使用的不同的权限要求。

又见

CREATE ROLE, ALTER ROLE, SET ROLE

DROP RULE

Name

DROP RULE -- 删除一个重写规则

Synopsis

```
DROP RULE [ IF EXISTS ] _name_ ON _table_name_ [ CASCADE | RESTRICT ]
```

描述

`DROP RULE` 删除一个规则。

参数

`IF EXISTS`

如果指定的规则不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的现存规则名称。

`_table_name_`

该规则应用的关系名字(可以有模式修饰)。

`CASCADE`

级联删除依赖于此规则的对象。

`RESTRICT`

如果有任何依赖对象，则拒绝删除此规则。这个是缺省。

例子

删除重写规则 `newrule` ：

```
DROP RULE newrule ON mytable;
```

兼容性

`DROP RULE`是一个PostgreSQL语言的扩展， 就像查询重写系统。

又见

CREATE RULE, ALTER RULE

DROP SCHEMA

Name

DROP SCHEMA -- 删除一个模式

Synopsis

```
DROP SCHEMA [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SCHEMA`从数据库中删除模式。

模式只能被它的所有者或者超级用户删除。请注意， 所有者即使没有拥有模式中任何对象也可以删除模式(以及模式中的所有对象)。

参数

`IF EXISTS`

如果指定的模式不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

模式的名字。

`CASCADE`

自动删除包含在模式中的对象(表、函数等等)。

`RESTRICT`

如果模式包含任何对象，则拒绝删除它。这个是缺省。

例子

从数据库中删除模式 `mystuff` 以及它包含的所有东西：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

`DROP SCHEMA`和 SQL 标准完全兼容，只不过标准只允许每条命令删除一个模式。 另外， `IF EXISTS` 选项是PostgreSQL扩展。

又见

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

Name

DROP SEQUENCE -- 删除一个序列

Synopsis

```
DROP SEQUENCE [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP SEQUENCE`从数据库中删除序列号生成器。只有其所有者或超级用户才能将其删除。

参数

`IF EXISTS`

如果指定的序列不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

序列名(可以有模式修饰)。

`CASCADE`

级联删除依赖序列的对象。

`RESTRICT`

如果存在任何依赖的对象，则拒绝删除序列。这个是缺省。

例子

从数据库中删除序列 `serial` ：

```
DROP SEQUENCE serial;
```

兼容性

`DROP SEQUENCE`遵循SQL标准，只是标准只允许每条命令删除一个序列。并且，`IF EXISTS`选项是PostgreSQL的扩展。

又见

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

Name

DROP SERVER -- 删除一个外部服务器描述符

Synopsis

```
DROP SERVER [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP SERVER` 删除一个现有外部服务器描述符。为了执行该命令， 当前用户必须是该服务器的所有者。

参数

`IF EXISTS`

如果指定的服务器不存在，那么发出一个notice而不是抛出一个错误。

`_name_`

一个既有服务器的名称。

`CASCADE`

级联删除依赖服务器的对象（比如用户映射）。

`RESTRICT`

若有任何对象依赖服务器则拒绝删除它。这个是默认。

例子

若存在则删除服务器 `foo`：

```
DROP SERVER IF EXISTS foo;
```

兼容性

`DROP SERVER`遵守ISO/IEC 9075-9 (SQL/MED)。 `IF EXISTS`子句是PostgreSQL扩展。

又见

CREATE SERVER, ALTER SERVER

DROP TABLE

Name

DROP TABLE -- 删除一个表

Synopsis

```
DROP TABLE [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TABLE` 从数据库中删除表或视图。只有表的者、 模式所有者和超级用户才能删除一个表。要清空而不是删除表，请使用 DELETE或TRUNCATE。

`DROP TABLE` 总是删除目标表上现有的任何索引、规则、触发器、约束。 但是，要删除一个有视图或者其它表用外键约束引用的表，必须声明 `CASCADE` 。 `CASCADE` 将删除引用的视图，但是如果是外键约束，那么就只删除外键约束， 而不是另外一个表。

参数

`IF EXISTS`

如果指定的表不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的现存表的名字(可以有模式修饰)。

`CASCADE`

级联删除依赖于表的对象(比如视图)。

`RESTRICT`

如果存在依赖对象，则拒绝删除该表。这个是缺省。

例子

删除 `films` 和 `distributors` 表：

```
DROP TABLE films, distributors;
```

兼容性

此命令兼容 SQL 标准。只不过标准只允许一条命令删除一个表。此外， `IF EXISTS` 选项是 PostgreSQL的扩展。

又见

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

Name

DROP TABLESPACE -- 删除一个表空间

Synopsis

```
DROP TABLESPACE [ IF EXISTS ] _name_
```

描述

`DROP TABLESPACE` 从系统里删除一个表空间。

一个表空间只能由其所有者或者超级用户删除。在删除一个表空间之前， 表空间里面不能有任何数据库对象。即使当前数据库里面已经没有任何对象在使用这个表空间了， 也有可能有其它的数据库对象存留在这个表空间里。同样，如果在任何活动会话的 temp_tablespaces中列出了该表空间， `DROP` 可能会因为临时文件存在于该表空间中而失败。

参数

`IF EXISTS`

如果指定的表空间不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

表空间的名字。

注意

`DROP TABLESPACE` 不能出现在事务块内部。

例子

从系统里删除表空间 `mystuff` ：

```
DROP TABLESPACE mystuff;
```

兼容性

`DROP TABLESPACE` 是PostgreSQL扩展。

又见

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

Name

DROP TEXT SEARCH CONFIGURATION -- 删除一个文本搜索配置

Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH CONFIGURATION` 删除既有文本搜索配置。 要执行这个命令，你必须是该配置的所有者。

参数

`IF EXISTS`

如果指定的文本搜索配置不存在，那么发出一个notice而不是抛出一个错误。

`_name_`

一个现有文本搜索配置的名称（可有模式修饰）。

`CASCADE`

级联删除依赖文本搜索配置的对象。

`RESTRICT`

若有任何对象依赖文本搜索配置则拒绝删除它。这是默认情况。

例子

删除文本搜索配置 `my_english`：

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

若有任何现有索引引用 `to_tsvector` 调用中的配置，该命令将不会成功。 添加 `CASCADE` 来删除这样的索引连带文本搜索配置。

兼容性

在SQL标准中没有 `DROP TEXT SEARCH CONFIGURATION` 语句。

又见

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

Name

DROP TEXT SEARCH DICTIONARY -- 删除一个文本搜索字典

Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH DICTIONARY` 删除一个文本搜索字典。 要执行这个命令，你必须是该字典的所有者。

参数

`IF EXISTS`

如果指定的文本搜索字典不存在，那么发出一个notice而不是抛出一个错误。

`_name_`

一个现有文本搜索字典的名称（可有模式修饰）。

`CASCADE`

级联删除依赖该文本搜索字典的对象。

`RESTRICT`

若有任何对象依赖文本搜索字典则拒绝删除它。这是默认情况。

例子

删除文本搜索字典 `english`：

```
DROP TEXT SEARCH DICTIONARY english;
```

若有使用该字典的任何现有文本搜索配置，该命令将不会成功。 添加 `CASCADE` 来删除字典连同文本搜索配置。

兼容性

在SQL标准中没有 `DROP TEXT SEARCH DICTIONARY` 语句。

又见

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

Name

DROP TEXT SEARCH PARSER -- 删除一个文本搜索解析器

Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH PARSER` 删除一个现有文本搜索解析器。你必须是超级用户才能使用此命令。

参数

`IF EXISTS`

如果指定的文本搜索解析器不存在，那么发出一个notice而不是抛出一个错误。

`_name_`

一个现有文本搜索解析器的名称（可有模式修饰）。

`CASCADE`

级联删除依赖该文本搜索解析器的对象。

`RESTRICT`

若有任何对象依赖文本搜索解析器则拒绝删除它。这是默认情况。

例子

删除文本搜索解析器 `my_parser`：

```
DROP TEXT SEARCH PARSER my_parser;
```

若有使用该解析器的任何现有文本搜索配置，该命令将不会成功。 添加 `CASCADE` 来删除配置连同解析器。

兼容性

在SQL标准中没有 `DROP TEXT SEARCH PARSER` 语句。

又见

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

Name

DROP TEXT SEARCH TEMPLATE -- 删除一个文本搜索模板

Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] _name_ [ CASCADE | RESTRICT ]
```

描述

`DROP TEXT SEARCH TEMPLATE` 删除一个现有文本搜索模板。 你必须是超级用户才能使用此命令。

参数

`IF EXISTS`

如果指定的文本搜索模板不存在，那么发出一个notice而不是抛出一个错误。

`_name_`

既有文本搜索模板的名称（可有模式修饰）。

`CASCADE`

级联删除依赖该文本搜索模板的对象。

`RESTRICT`

若有任何对象依赖文本搜索模板则拒绝删除它。这是默认的情况。

例子

删除文本搜索模板 `thesaurus`：

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

若有任何使用模板的既有文本搜索字典则该命令将不会成功。 添加 `CASCADE` 来删除此类字典以及模板。

兼容性

在SQL标准中没有 `DROP TEXT SEARCH TEMPLATE` 语句。

又见

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRIGGER

Name

DROP TRIGGER -- 删除一个触发器

Synopsis

```
DROP TRIGGER [ IF EXISTS ] _name_ ON _table_name_ [ CASCADE | RESTRICT ]
```

描述

`DROP TRIGGER` 删除一个现存触发器的定义。要执行这个命令， 当前用户必须是定义触发器所在的表的所有者。

参数

`IF EXISTS`

如果指定的触发器不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的触发器名。

`_table_name_`

触发器定义所在的表的名称(可以有模式修饰),

`CASCADE`

级联删除依赖此触发器的对象。

`RESTRICT`

如果有任何依赖对象存在，那么拒绝删除。这个是缺省。

例子

删除 `films` 表的 `if_dist_exists` 触发器：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

PostgreSQL里的 `DROP TRIGGER` 语句和 SQL 标准不兼容。在 SQL 标准里，触发器名字不是表所局部拥有的， 所以命令只是简单的 `DROP TRIGGER` `_name_` 。

又见

CREATE TRIGGER

DROP TYPE

Name

DROP TYPE -- 删除一个数据类型

Synopsis

```
DROP TYPE [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP TYPE` 删除一个用户定义的数据类型。只有类型所有者才可以删除它。

参数

`IF EXISTS`

如果指定的类型不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的类型名(可以有模式修饰)。

`CASCADE`

级联删除依赖该类型的对象(比如字段、函数、操作符等等)

`RESTRICT`

如果有依赖对象，则拒绝删除该类型。这个是缺省。

例子

删除 `box` 类型：

```
DROP TYPE box;
```

兼容性

这条命令类似于 SQL 标准里对应的命令，不过，`IF EXISTS` 选项是PostgreSQL的扩展。但是要注意，PostgreSQL 里的大多数 `CREATE TYPE` 命令和数据类型扩展机制是和 SQL 标准里不同的。

又见

ALTER TYPE, CREATE TYPE

DROP USER

Name

DROP USER -- 删除一个数据库角色

Synopsis

```
DROP USER [ IF EXISTS ] _name_ [, ...]
```

描述

`DROP USER` 现在是DROP ROLE的别名。

兼容性

`DROP USER` 语句是一个PostgreSQL的扩展。 SQL 标准把用户的定义交给具体实现处理。

又见

DROP ROLE

DROP USER MAPPING

Name

DROP USER MAPPING -- 删除用户的外部服务器映射

Synopsis

```
DROP USER MAPPING [ IF EXISTS ] FOR { _user_name_ | USER | CURRENT_USER | PUBLIC } SERVER
```

描述

`DROP USER MAPPING` 从外部服务器删除一个已存在的用户映射。

外部服务器的属主可以删除该服务器上任意用户的用户映射。 另外，如果 `USAGE` 服务器上的权限已经被授权给该用户，该用户可以删除其同名的用户映射

参数

`IF EXISTS`

当用户映射不存在的时候抛错。在此情况下发提示。

`_user_name_`

映射的用户名。 `CURRENT_USER` 和 `USER` 匹配当前用户的名称。 `PUBLIC` 用于匹配系统中的所有目前和未来用户名。

`_server_name_`

用户映射的服务名。

示例

当服务 `foo` 存在时，删除一个用户映射 `bob` ：

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

兼容性

`DROP USER MAPPING` 符合 ISO/IEC 9075-9 (SQL/MED)。 `IF EXISTS` 子句是 PostgreSQL 的扩展选项。

另请参见

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

Name

DROP VIEW -- 删除一个视图

Synopsis

```
DROP VIEW [ IF EXISTS ] _name_ [, ...] [ CASCADE | RESTRICT ]
```

描述

`DROP VIEW` 从数据库中删除一个现存的视图。执行这条命令必须是视图的所有者。

参数

`IF EXISTS`

如果指定的视图不存在，那么发出一个 notice 而不是抛出一个错误。

`_name_`

要删除的视图名称(可以有模式修饰)。

`CASCADE`

级联删除依赖此视图的对象(比如其它视图)。

`RESTRICT`

如果有依赖对象存在，则拒绝删除此视图。这个是缺省。

例子

下面命令将删除 `kinds` 视图：

```
DROP VIEW kinds;
```

兼容性

这条命令遵循 SQL 标准。只是标准只允许一条命令删除一个视图。此外， `IF EXISTS` 选项是PostgreSQL的扩展。

又见

ALTER VIEW, CREATE VIEW

END

Name

END -- 提交当前事务

Synopsis

```
END [ WORK | TRANSACTION ]
```

描述

`END` 提交当前事务。 所有当前事务做的修改都可被其它事务看到并且保证崩溃情况下的持续性。 它是一个PostgreSQL的扩展，等效于COMMIT。

参数

`WORK``TRANSACTION`

可选关键字，没有作用。

注意

用ROLLBACK退出事务。

在一个事务块之外发出 `END` 不会有什么损害，但是它会生成一个警告信息。

例子

提交当前事务，令所有改变生效：

```
END;
```

兼容性

`END` 是PostgreSQL的扩展， 提供与COMMIT相同的功能，后者是 SQL 标准声明的语句。

又见

BEGIN, COMMIT, ROLLBACK

EXECUTE

Name

EXECUTE -- 执行一个预备语句

Synopsis

```
EXECUTE _name_ [ ( _parameter_ [, ...] ) ]
```

描述

`EXECUTE` 执行一个前面准备好的预备语句。因为一个预备语句只在会话的生命期里存在， 那么预备语句必须是在当前会话的前些时候用 `PREPARE` 语句创建的。

如果创建预备语句的 `PREPARE` 语句声明了一些参数， 那么传递给 `EXECUTE` 语句的必须是一个兼容的参数集，否则就会生成一个错误。 请注意(和函数不同)，预备语句不会基于参数的类型或者参数个数重载：在一次数据库会话过程中， 预备语句的名字必须是唯一的。

有关创建和使用预备语句的更多信息，请参阅PREPARE。

参数

`_name_`

要执行的预备语句的名字。

`_parameter_`

给预备语句的一个参数的具体数值。 它必须是一个生成与创建这个预备语句时指定参数的数据类型相兼容的值的表达式。

输出

`EXECUTE` 返回的命令标签是预备语句的命令标签，不是 `EXECUTE` 的。

Examples

例子在PREPARE文档的*Examples*小节给出。

兼容性

SQL 标准包括一个 `EXECUTE` 语句，但它只用于嵌入式 SQL 。 `EXECUTE` 实现的 `EXECUTE` 的语法也略微不同。

又见

DEALLOCATE, PREPARE

EXPLAIN

Name

EXPLAIN -- 显示一个语句的执行规划

Synopsis

```
EXPLAIN [ ( _option_ [, ...] ) ] _statement_
EXPLAIN [ ANALYZE ] [ VERBOSE ] _statement_

这里的 `_option_`可以是下列之一:

    ANALYZE [ _boolean_ ]
    VERBOSE [ _boolean_ ]
    COSTS [ _boolean_ ]
    BUFFERS [ _boolean_ ]
    TIMING [ _boolean_ ]
    FORMAT { TEXT | XML | JSON | YAML }
```

描述

这条命令显示PostgreSQL规划器为所提供的语句生成的执行规划。 执行规划显示语句引用的表是如何被扫描的(简单的顺序扫描，还是索扫描)，并且如果引用了多个表， 采用了什么样的连接算法从每个输入的表中取出所需要的记录。

显示出来的最关键的部分是预计的语句执行开销，这就是规划器对运行该语句所需时间的估计 (以任意的开销单位计量，但是通常意味着磁盘页面存取)。实际上显示了两个数字： 返回第一行记录前的启动开销，和返回所有记录的总开销。对于大多数查询而言，关心的是总开销， 但是，在某些环境下，比如一个 `EXISTS` 子查询里， 规划器将选择最小启动开销而不是最小总开销(因为执行器在获取一条记录后总是要停下来)。 同样，如果你用一条 `LIMIT` 子句限制返回的记录数， 规划器会在最终的开销上做一个合理的插值以计算哪个规划开销最省。

`ANALYZE` 选项导致查询被实际执行，而不仅仅是规划。显示中加入了实际的运行时间统计，包括在每个规划节点内部花掉的总时间(以毫秒计)和它实际返回的行数。 这些数据对搜索该规划器的预期是否和现实相近很有帮助。

> **Important:** 要记住的是查询实际上在使用`ANALYZE`选项的时候是执行的。 尽管`EXPLAIN`会抛弃任何`SELECT`返回的输出， 但是其它查询的副作用还是一样会发生的。如果你在`INSERT`，`UPDATE`，`DELETE`，`CREATE TABLE AS`，`EXECUTE` 语句里使用`EXPLAIN ANALYZE`而且还不想让查询影响数据，可以用下面的方法：
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

只能声明`ANALYZE`和`VERBOSE`选项，并且只能以那种顺序， 不能将选项列表放在圆括号中。 PostgreSQL 9.0之前， 只支持不用圆括号的语法。人们希望只在圆括号语法中支持所有新的选项。

参数

`ANALYZE`

执行命令并显示实际运行时间和其他统计。这个参数缺省为`FALSE`。

`VERBOSE`

显示关于规划的额外的信息。特别的包括规划树上的每个节点的输出字段列表， 模式修饰表和函数名，表达式中的标签变量总是和他们的范围表别名在一起， 并且总是打印统计数据中显示的每个触发器的名字。这个参数缺省为`FALSE`。

`COSTS`

包括每个规划节点的估计启动成本和总成本的信息，也包括估计行数和估计的每行的宽度。这个参数缺省为`TRUE`。

`BUFFERS`

包含缓冲区使用的信息。特别的，包括共享块命中、读、脏和写的次数，本地块命中、读、脏和写的次数， 临时块读和写的次数。命中意味着避免了读，因为块在需要时已经在缓存中发现了。 共享块包含普通表和索引的数据；本地块包含临时表和索引的数据；而临时块包含用于排序、哈希、 物化规划节点和相似情况的短期工作数据。脏块的数量表示这个查询改变的先前未更改的块的数量； 写块的数量表示在查询处理的时候被这个后端驱逐出缓存的先前脏了的块的数量。 高级节点显示的块的数量包含所有它的子节点使用的块的数量。在文本格式中，只打印非零值。 这个参数可能只在`ANALYZE`也启用的时候使用。它的缺省为`FALSE`。

`TIMING`

在输出中包含实际启动时间和每个节点花费的时间。重复读系统块的总开销会在某些系统上显著的减缓查询的速度， 所以当需要只有实际行被计算，并且没有准确时间时，设置这个参数为`FALSE`会很有用。 即使是用这个选项关闭了节点级别的时间，也测量整个语句的运行时间。 这个参数可能只在`ANALYZE`也启用的时候使用。它缺省为`TRUE`。

`FORMAT`

声明输出格式，可以为TEXT, XML, JSON 或 YAML。非文本的输出包含文本输出格式相同的信息， 但是更容易被程序解析。这个参数缺省为 `TEXT` 。

`_boolean_`

声明选中的选项打开或者关闭。可以用 `TRUE` , `ON` 或 `1` 启用这个选项，用 `FALSE` , `OFF` 或 `0` 禁用这个选项。在假设为 `TRUE` 的情况下， `_boolean_` 值也可以忽略，

`_statement_`

你想要查看执行规划的任何 `SELECT` , `INSERT` , `UPDATE` , `DELETE` , `VALUES` , `EXECUTE` , `DECLARE` , 或 `CREATE TABLE AS` 语句之一。

输出

命令的结果是从 `_statement_` 选择的规划的文字描述， 可选的有执行统计数据的注释。Section 14.1描述提供的信息。

注意

为了让PostgreSQL查询规划器在优化查询的时候做出合理的判断， `pg_statistic` 数据应该为所有用于查询的表更新。通常autovacuum daemon 会自动注意这些。但是如果一个表最近在内容上有大量的更改，你可能需要手动ANALYZE 而不是等待autovacuum赶上变化。

为了测量运行时在执行规划中每个节点的开销， `EXPLAIN ANALYZE` 的当前应用增加查询执行开销的性能分析。结果，在一个查询上运行 `EXPLAIN ANALYZE` 有时会比普通查询明显的花费更多的时间。超支的数量依赖于查询的本质和使用的平台。 最坏的情况发生在他们本身的规划节点时每个执行需要非常少的时间， 并且在操作系统相当慢的机器上需要获得时刻。

例子

显示一个对只有一个 `integer` 列和 10000 行表的简单查询的查询规划：

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

相同的查询用JSON输出格式：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

如果存在一个索引，并且使用一个可应用索引的 `WHERE` 条件的查询， `EXPLAIN` 会显示不同的规划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

相同的查询，但是用YAML格式：

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

XML格式留给读者做练习。

这是相同的查询关闭成本估算：

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

下面是一个使用了聚集函数的查询的查询规划的例子：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                             QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

下面是一个使用 `EXPLAIN EXECUTE` 显示一个已预编写的查询规划的例子：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                       QUERY PLAN
------------------------------------------------------------------------------------
 HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops
   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8) (actual t
         Index Cond: ((id > $1) AND (id < $2))
 Total runtime: 0.851 ms
(4 rows)
```

注意这里显示的数字，依赖于所包含的表的实际内容。还请注意该数字甚至还有选择的查询策略都有可能在各个 PostgreSQL版本之间不同，因为规划器在不断改进。另外，`ANALYZE` 命令使用随机的采样来估计数据统计；因此，一次新的 `ANALYZE` 运行之后开销估计可能会变化，即使数据的实际分布没有改变也这样。

兼容性

在 SQL 标准中没有 `EXPLAIN` 语句。

又见

ANALYZE

FETCH

Name

FETCH -- 用游标从查询中抓取行

Synopsis

```
FETCH [ _direction_ [ FROM | IN ] ] _cursor_name_

这里的 `_direction_` 可以为空或下列之一:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE _count_
    RELATIVE _count_
    _count_
    ALL
    FORWARD
    FORWARD _count_
    FORWARD ALL
    BACKWARD
    BACKWARD _count_
    BACKWARD ALL
```

描述

`FETCH` 使用游标检索行。

一个游标有一个由 `FETCH` 使用的相关联的位置。游标的位置可以在查询结果的第一行之前，或者在结果中的任意行，或者在结果的最后一行之后。刚创建完之后，游标是放在第一行之前的。 在抓取了一些行之后，游标放在检索到的最后一行上。如果 `FETCH` 抓完了所有可用行， 那么它就停在最后一行后面，或者在反向抓取的情况下是停在第一行前面。 `FETCH ALL` 或 `FETCH BACKWARD ALL` 将总是把游标的位置放在最后一行或者在第一行前面。

`NEXT` , `PRIOR` , `FIRST` , `LAST` , `ABSOLUTE` , `RELATIVE` 形式在恰当地移动游标之后抓取一个行。 如果没有数据行了，那么返回一个空的结果，此时游标就会停在查询结果的最后一行之后或者第一行之前。

`FORWARD` 和 `BACKWARD` 形式在向前或者向后移动的过程中抓取指定的行数， 然后把游标定位在最后返回的行上；或者是，如果 `_count_` 大于可用的行数，则在所有行之前或之后。

`RELATIVE 0` , `FORWARD 0` , `BACKWARD 0` 都要求在不移动游标的前提下抓取当前行，也就是重新抓取最近刚刚抓取过的行。 除非游标定位在第一行之前或者最后一行之后，这个动作都应该成功， 而在那两种情况下，不返回任何行。

> **Note:** 本页描述在SQL命令级别的游标的使用。如果你尝试在PL/pgSQL 里使用游标，那么规则是不同的，参阅Section 40.7。

参数

`_direction_`

`_direction_` 定义抓取的方向和抓取的行数。 它可以是下述之一：

`NEXT`

抓取下一行(缺省)。

`PRIOR`

抓取前面一行。

`FIRST`

抓取查询的第一行(和 `ABSOLUTE 1` 相同)。

`LAST`

抓取查询的最后一行(和 `ABSOLUTE -1` 相同)。

`ABSOLUTE` `_count_`

抓取查询中第 `_count_` 行，或者如果 `_count_` 为负就从查询结果末尾抓取第 `abs(``_count_` )行。 如果 `_count_` 超出了范围， 那么定位在第一行之前和最后一行之后的位置；特别是 `ABSOLUTE 0` 定位在第一行之前。

`RELATIVE` `_count_`

抓取随后的第 `_count_` 行，或者如果 `_count_` 为负就抓取前面的第 `abs(``_count_` )行。 如果有数据的话， `RELATIVE 0` 重新抓取当前行。

`_count_`

抓取随后的 `_count_` 行(和 `FORWARD` `_count_` 一样)。

`ALL`

抓取所有剩余的行(和 `FORWARD ALL` 一样)。

`FORWARD`

抓取下一行(和 `NEXT` 一样)。

`FORWARD` `_count_`

抓取随后的 `_count_` 行。 `FORWARD 0` 重新抓取当前行。

`FORWARD ALL`

抓取所有剩余行。

`BACKWARD`

抓取前面一行(和 `PRIOR` 一样)。

`BACKWARD` `_count_`

抓取前面的 `_count_` 行(向后扫描)。 `BACKWARD 0` 重新抓取当前行。

`BACKWARD ALL`

抓取所有前面的行(向后扫描)。

`_count_`

`_count_` 可能是一个有符号的整数常量， 决定要抓取的行数和方向。对于 `FORWARD` 和 `BACKWARD` 的情况， 声明一个带负号的 `_count_` 等效于改变 `FORWARD` 和 `BACKWARD` 的方向。

`_cursor_name_`

一个打开的游标的名称。

输出

成功完成时，一个 `FETCH` 命令返回一个形如下面的标记

```
FETCH _count_
```

这里的 `_count_` 是抓取的行数(可能是零)。 请注意在psql里，命令标签实际上不会显示， 因为psql用抓取的行数代替了。

注意

如果你想使用 `FETCH NEXT` 之外的任何 `FETCH` 变种，或者是带负数计数的 `FETCH FORWARD` ，那么定义游标的时候应该带上 `SCROLL` 选项。 对于简单的查询，PostgreSQL会允许那些没有带 `SCROLL` 选项定义的游标也可以反向抓取，但是最好不要依赖这个行为。如果游标定义了 `NO SCROLL` ，那么不允许反向抓取。

`ABSOLUTE` 抓取不会比用相对位移移动到需要的数据行更快： 因为下层的实现必须遍历所有中间的行。负数的绝对抓取甚至更糟糕： 查询必须一直读到结尾才能找到最后一行，然后从那里开始反向遍历。不过， 回退到查询开头(比如 `FETCH ABSOLUTE 0`)很快。

使用DECLARE语句定义一个游标。MOVE 语句改变游标位置而不检索数据。

例子

下面的例子用一个游标跨过一个表。

```
BEGIN WORK;

-- 建立一个游标:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- 抓取头5行到游标liahona里:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- 抓取前面行:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- 关闭游标并提交事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准定义的 `FETCH` 只用于嵌入式环境下。这里描述的 `FETCH` 变种是把结果数据像 `SELECT` 结果那样返回， 而不是把它放在属主变量里。除了这点之外， `FETCH` 和 SQL 标准完全向前兼容。

涉及 `FORWARD` 和 `BACKWARD` 的 `FETCH` 形式， 包括 `FETCH` `_count_` 和 `FETCH ALL` 形式(此时 `FORWARD` 是隐含的)， 是PostgreSQL的扩展。

SQL 标准只允许游标前面有 `FROM` ，可选的 `IN` ，或者把它们都省去，是一个扩展。

又见

CLOSE, DECLARE, MOVE

GRANT

Name

GRANT -- 赋予访问权限

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] _table_name_ [, ...]
         | ALL TABLES IN SCHEMA _schema_name_ [, ...] }
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( _column_name_ [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( _column_name_ [, ...] ) }
    ON [ TABLE ] _table_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE _sequence_name_ [, ...]
         | ALL SEQUENCES IN SCHEMA _schema_name_ [, ...] }
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE _database_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN _domain_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER _fdw_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER _server_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION _function_name_ ( [ [ _argmode_ ] [ _arg_name_ ] _arg_type_ [, ...] ] )
         | ALL FUNCTIONS IN SCHEMA _schema_name_ [, ...] }
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE _lang_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT _loid_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA _schema_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE _tablespace_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE _type_name_ [, ...]
    TO { [ GROUP ] _role_name_ | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT _role_name_ [, ...] TO _role_name_ [, ...] [ WITH ADMIN OPTION ]
```

描述

`GRANT` 命令有两个基本变种：一个变种是给数据库对象 (表、字段、视图、外部表、序列、数据库、外部数据封装器、外部服务器、函数、过程语言、 模式、表空间)赋予权限；一个变种是赋予一个角色中的成员关系。这些变种在很多方面都非常类似， 但是它们之间的区别也有足够理由来分开描述。

在数据库对象上的 GRANT

这个变种的 `GRANT` 命令在数据库对象上给一个或多个角色授予特定的权限。 这些权限追加到已经授予的权限上。

也有一个选项赋予一个或多个模式内的相同类型的所有对象权限。这个功能当前只支持表，序列和函数（但请注意 `ALL TABLES` 被认为包含视图和外部表）。

关键字 `PUBLIC` 表示该权限要赋予所有角色，包括那些以后可能创建的用户。 `PUBLIC` 可以看做是一个隐含定义好的组，它总是包括所有角色。 任何特定的角色都将拥有直接赋予他/它的权限，加上他/它所处的任何组， 以及再加上赋予 `PUBLIC` 的权限的总和。

如果声明了 `WITH GRANT OPTION` ，那么权限的接收者也可以将此权限赋予他人， 否则就不能授权他人。这个选项不能赋予 `PUBLIC` 。

对于对象的所有者(通常就是创建者)而言，没有什么权限需要赋予，因为所有者缺省就持有所有权限。 不过，所有者出于安全考虑可以选择废弃一些他自己的权限。

删除一个对象的权力，或者是任意修改它的权力都不是可赋予的权限；它是创建者固有的，并且不能赋予或撤销。（然而，一个类似的影响可以通过赋予或撤销拥有这个对象的角色的成员关系来获得； 参阅下文。）所有者也隐含地拥有该对象的所有授权选项。

PostgreSQL给 `PUBLIC` 对象的某些类型赋予缺省的权限。在表、字段、 模式或表空间上缺省不会赋予 `PUBLIC` 权限。对于其他类型， 赋予 `PUBLIC` 以下缺省的权限：数据库为 `CONNECT` 和 `CREATE TEMP TABLE` 权限；函数为 `EXECUTE` 权限； 语言为 `USAGE` 权限。对象所有者当然可以 `REVOKE` 缺省的或明确赋予的权限。出于最大安全性考虑，在创建该对象的同一个事务中发出 `REVOKE` 就不会打开给别的用户使用该对象的窗口。同样， 这些缺省初始化权限设置可以使用ALTER DEFAULT PRIVILEGES命令改变。

可能的权限有：

SELECT

允许对声明的表、视图、序列SELECT任意字段或指定字段列表。 还允许做COPY的源。这个权限也需要引用UPDATE 或DELETE现存的字段值。对于序列而言，这个权限还允许使用 `currval` 函数。对于大对象，这个权限允许读对象。

INSERT

允许向声明的表INSERT一个新行。如果列出了指定的字段， 那么只有列出的字段被指派给 `INSERT` 命令（其他字段因此接受缺省的值）。 同时还允许做COPY。

UPDATE

允许对声明的表中任意字段或指定的字段列表做UPDATE。 实际上，任何重要的 `UPDATE` 命令也需要 `SELECT` 权限， 因为必须引用表字段决定要更新哪个行，和/或为字段计算新值。 `SELECT ... FOR UPDATE` 和 `SELECT ... FOR SHARE` 也至少在一个字段上要求这个权限，除了 `SELECT` 权限之外。比如， 这个权限允许使用 `nextval` 和 `setval` 函数。 对于大对象，这个权限允许写或截断对象。

DELETE

允许从声明的表中DELETE行。实际上，任何重要的 `DELETE` 命令也需要 `SELECT` 权限，因为必须引用表字段决定要删除哪个行。

TRUNCATE

允许在指定的表上TRUNCATE。

REFERENCES

要创建一个外键约束，你必须在参考字段和被参考字段上都拥有这个权限。 该权限可能赋予了表的所有字段，或只是赋予了指定的字段。

TRIGGER

允许在声明表上创建触发器(参见CREATE TRIGGER语句)。

CREATE

对于数据库，允许在该数据库里创建新的模式。

对于模式，允许在该模式中创建新的对象。要重命名一个现有对象， 你必需拥有该对象并且对包含该对象的模式拥有这个权限。

对于表空间，允许在其中创建表，索引和临时文件，以及允许创建数据库的时候把该表空间指定为其缺省表空间。 请注意，撤销这个权限不会改变现有对象的存放位置。

CONNECT

允许用户连接到指定的数据库。该权限将在连接启动时检查 (除了检查 `pg_hba.conf` 中的任何限制之外)。

TEMPORARY TEMP

允许在使用指定数据库的时候创建临时表

EXECUTE

允许使用指定的函数并且可以使用任何利用这些函数实现的操作符。 这是适用于函数的唯一权限。该语法同样适用于聚集函数。

USAGE

对于过程语言，允许使用指定过程语言创建该语言的函数。这是适用于过程语言的唯一权限。

对于模式，允许访问包含在指定模式中的对象(假设该对象的所有权要求同样也设置了)。 最终这些就允许了权限接受者"查询"模式中的对象。没有这个权限仍然可以看见这些对象的名字(比如通过查询系统视图)。同样，撤销该权限之后，现有的后端可能有在查找之前就执行了的语句， 因此这不是一个很安全的限制对象访问的方法。

对于序列，该权限允许使用 `currval` 和 `nextval` 函数。

对于类型和域，该权限允许在创建表、函数和其他模式对象时使用类型或域。 （请注意，它不控制类型的一般"使用"，例如显示在查询中的该类型的值。 它只防止依赖于该类型的对象被创建。该权限的主要目的是控制哪个用户在类型上创建依赖， 该依赖会防止用户稍后改变类型。）

对于外部数据封装器，该权限使权限接受者能够使用那个外部数据封装器创建新的服务器。

对于服务器，该权限使权限接受者能够使用该服务器创建外部表，并且也能创建、 修改或删除他自己的用户的与该服务器相关的用户映射。

ALL PRIVILEGES

一次性给予所有可以赋予的权限。 `PRIVILEGES` 关键字在 PostgreSQL里是可选的，但是严格的 SQL 要求有这个关键字。

其它命令要求的权限都在相应的命令的参考页上列出。

角色上的 GRANT

这个变种的 `GRANT` 命令把一个角色的成员关系赋予一个或多个其它角色。 角色里的成员关系很重要，因为它会将赋予该角色的权限传播给所有该角色的成员。

如果声明了 `WITH ADMIN OPTION` ，那么该成员随后就可以将角色的成员关系赋予其它角色， 以及撤销其它角色的成员关系。如果没有 admin 选项，普通用户就不能这么做。不过， 数据库超级用户可以给任何人赋与或者撤销任何角色的任何成员关系。拥有 `CREATEROLE` 权限的角色可以赋予或者撤销任何非超级用户角色的成员关系。

与权限不同，角色的成员关系不能被赋予 `PUBLIC` 。需要注意的是， 这种形式的命令不允许使用无意义的 `GROUP` 关键字。

注意

REVOKE命令用于删除访问权限。

自PostgreSQL 8.1以来，用户和组的概念统一为角色。 因此不再需要使用关键字 `GROUP` 来确定接受者是一个用户还是一个组。 仍然允许在命令中出现 `GROUP` ，但这是一个噪声字。

一个用户可以他有相应权限的字段上执行 `SELECT` ，`INSERT` 等，该权限是在指定字段或整个表上。 在表级别授予权限然后从一个字段上撤销该权限将不会做你希望发生的事情：表级别的授权不受字段级别操作的影响。

如果非对象所有者企图在对象上 `GRANT` 权限，而该用户没有该对象上指定的权限， 那么命令将立即失败。只要有某些可用的权限，该命令就会继续，但是它只授予那些该用户有授权选项的权限。 如果没有可用的授权选项，那么 `GRANT ALL PRIVILEGES` 形式将发出一个警告信息， 其它命令形式将发出在命令中提到的、但是没有授权选项的那些权限相关的警告信息。 这些语句原则上也适用于对象所有者，但是因为所有者总是被认为拥有所有授权选项， 所以这种情况永远不会发生在所有者身上。

要注意数据库超级用户可以访问所有对象，而不会受对象的权限设置影响。 这个特点类似Unix 系统的 `root` 的权限。和 `root` 一样，除了必要的情况， 总是以超级用户身份进行操作是不明智的做法。

如果一个超级用户选择发出一个 `GRANT` 或 `REVOKE` 命令， 那么这条命令将是以被影响对象的所有者的形式执行的。特别是， 通过这种方法赋与的权限将显得好像是由对象所有者赋与的。对于角色成员关系， 成员关系的赋与就会像是通过包含角色自己赋与的一样。

`GRANT` 和 `REVOKE` 也可以不由被影响对象的所有者来执行， 而是由拥有该对象的角色的一个成员来执行，或者是一个在该对象上持有 `WITH GRANT OPTION` 权限的角色的成员。在这种情况下， 该权限将被纪录为是由实际拥有该对象或者持有 `WITH GRANT OPTION` 权限的对象赋与的。比如，如果表 `t1` 被角色 `g1` 拥有， 并且 `u1` 是 `g1` 的一个成员，然后 `u1` 可以把 `t1` 的权限赋予 `u2` ，但是这些权限将表现为是由 `g1` 直接赋予的。 任何 `g1` 角色的成员都可以在之后撤销这些权限。

如果执行 `GRANT` 的角色所持有的所需权限是通过角色成员关系间接获得的， 那么究竟是那个角色将被纪录为赋予权限的角色就是未知的。在这种情况下， 最好的方法是使用 `SET ROLE` 成为你想执行 `GRANT` 命令的指定角色。

在表上赋予的权限不会自动传播到该表使用的序列上，包括 `SERIAL` 字段上的序列。 必须单独设置序列的权限。

使用psql的 `\dp` 命令获取表和字段的现存权限的有关的信息。例如：

```
=> \dp mytable
                              Access privileges
 Schema |  Name   | Type  |   Access privileges    | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

`\dp` 显示的条目解释如下：

```
rolename=xxxx -- 赋予一个角色的权限
        =xxxx -- 赋予 PUBLIC 的权限

            r -- SELECT ("读")
            w -- UPDATE ("写")
            a -- INSERT ("追加")
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
            X -- EXECUTE
            U -- USAGE
            C -- CREATE
            c -- CONNECT
            T -- TEMPORARY
      arwdDxt -- ALL PRIVILEGES (用于表，用于其他对象时不同)
            * -- 给前面权限的授权选项

        /yyyy -- 授出这个权限的角色
```

用户 `miriam` 在建完 `mytable` 表之后再做下面的语句，就可以得到上面例子的结果：

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

对于非表的对象，可以使用 `\d` 命令显示他们的权限。

如果一个给定的对象的"Access privileges"字段是空的，这意味着该对象有缺省权限 (也就是说，它的权限字段是 NULL)。缺省权限总是包括所有者的所有权限，以及根据对象的不同，可能包含一些给 `PUBLIC` 的权限。对象上第一个 `GRANT` 或 `REVOKE` 将实例化这个缺省权限(比如，产生 `{miriam=arwdDxt/miriam}`) 然后根据每次特定的需求修改它。 相似的，"Column access privileges"中的记录只为非缺省权限字段。（请注意：为了这个目的， "default privileges"总是意味着该对象类型的内建缺省权限。受 `ALTER DEFAULT PRIVILEGES` 命令影响权限的对象总是和一个包括 `ALTER` 影响的明确权限记录一起显示。）

请注意所有者的隐含授权选项没有在显示出来的访问权限里标记出来。 只有在授权选项明确地授予某人之后，才会显示一个 `*` 。

例子

把表 `films` 的插入权限赋予所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

赋予用户 `manuel` 对视图 `kinds` 的所有权限：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

请注意，如果上面的命令由超级用户或者 `kinds` 的所有者执行，那么它实际上会赋予所有权限， 如果由其他人执行，那么它会赋予这个"其他人"拥有授权选项的所有权限。

把角色 `admins` 的成员关系赋与用户 `joe` ：

```
GRANT admins TO joe;
```

兼容性

根据 SQL 标准，在 `ALL PRIVILEGES` 里的 `PRIVILEGES` 关键字是必须的。 SQL 标准不支持在一条命令里对多个表设置权限。

PostgreSQL允许一个对象所有者撤销它自己的普通权限：比如， 一个表所有者可以让自己对这个表是只读的，方法是撤销自己的 `INSERT` , `UPDATE` , `DELETE` , 和 `TRUNCATE` 权限。根据 SQL 标准， 这是不可能的。原因是PostgreSQL把所有者的权限当作由所有者给自己赋予的； 因此也可以撤销他们。在 SQL 标准里，所有者的权限是假设为"_SYSTEM"实体赋予的。 因为所有者不是"_SYSTEM"，所以他不能撤销这些权限。

根据 SQL 标准，授权选项可以授予 `PUBLIC` ；PostgreSQL 只支持授予授权选项给角色。

SQL 标准对其它类型的对象提供了一个 `USAGE` 权限：字符集、校勘、转换。

在SQL标准中，序列只有一个 `USAGE` 权限，该权限控制 `NEXT VALUE FOR` 表达式的使用，该表达式相当于PostgreSQL中的 `nextval` 函数。序列权限 `SELECT` 和 `UPDATE` 是PostgreSQL的扩展。序列 `USAGE` 权限应用到 `currval` 函数也是一个PostgreSQL扩展（就像函数本身）。

在数据库、表空间、模式、语言上的权限是PostgreSQL扩展。

又见

REVOKE, ALTER DEFAULT PRIVILEGES

INSERT

Name

INSERT -- 在表中创建新行

Synopsis

```
[ WITH [ RECURSIVE ] _with_query_ [, ...] ]
INSERT INTO _table_name_ [ ( _column_name_ [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { _expression_ | DEFAULT } [, ...] ) [, ...] | _query_ }
    [ RETURNING * | _output_expression_ [ [ AS ] _output_name_ ] [, ...] ]
```

描述

`INSERT` 向表中插入新行。可以一次插入用值表达式声明的一行或多行， 或者一个查询结果表现出来的零行或多个行。

目标列表中的列/字段可以按任何顺序排列。如果完全没有列出任何字段名，那么缺省是全部字段， 顺序是按照表声明的时候的顺序；如果 `VALUES` 子句或者 `_query_` 里面只提供了 `_N_` 个字段，那么就是头 `_N_` 个字段。 `VALUES` 子句或者 `_query_` 提供的数值是以从左到右的方式与明确或者隐含的字段列表关联的。

每个没有在明确或者隐含的字段列表中出现的字段都将填充缺省值， 如果有声明的缺省值则用声明的那个，如果没有则用 NULL 。

如果每个字段的表达式不是正确的数据类型，系统将试图进行自动的类型转换。

可选的 `RETURNING` 子句将导致 `INSERT` 计算和返回实际插入的行， 它主要用于获取缺省的计算值(比如序列值)，不过，任何使用该表字段的表达式都是允许的。 `RETURNING` 列表的语法都与 `SELECT` 的输出列表相同。

要想向表中插入数据，你必须有 `INSERT` 权限。如果指定了一个字段列表， 那么只需要在列表字段上的 `INSERT` 权限。使用 `RETURNING` 子句需要要在 `RETURNING` 中提到的所有字段上的 `SELECT` 权限。 如果你使用了 `_query_` 子句插入来自查询里的数据行， 你还需要拥有在查询里使用的表或字段的 `SELECT` 权限。

参数

`_with_query_`

`WITH` 子句允许声明一个或多个可以在 `INSERT` 查询中通过名字引用的子查询。参阅Section 7.8和 SELECT获取详细信息。

`_query_` (`SELECT` 语句) 也包含一个 `WITH` 子句是可能的。在这种情况下，两套 `_with_query_` 都可以在 `_query_` 中引用，但是第二个优先，因为它嵌套更紧密。

`_table_name_`

现存表的名称(可以有模式修饰)。

`_column_name_`

通过 `_table_name_` 命名的表中的字段名。必要时， 字段名可以有子字段名或者数组下标修饰。向一个复合类型中的某些字段插入数据的话， 其它字段将是 NULL 。

`DEFAULT VALUES`

所有字段都会用它们的缺省值填充。

`_expression_`

赋予对应的 column 的一个有效表达式或值。

`DEFAULT`

对应的 column 将被它的缺省值填充。

`_query_`

一个查询(`SELECT` 语句)，它提供插入的数据行。请参考SELECT 语句获取语法描述。

`_output_expression_`

`INSERT` 命令在每一行都被插入之后用于计算输出结果的表达式。该表达式可以使用 通过 `_table_name_` 命名的表的的任意字段。 可以使用 `*` 返回被插入行的所有字段。

`_output_name_`

一个字段的输出名称。

输出

成功完成后，一条 `INSERT` 命令返回一个下面形式的命令标签

```
INSERT _oid_ _count_
```

`_count_` 是插入的行数。如果 `_count_` 正好是一，并且目标表有 OID， 那么 `_oid_` 是赋予插入行的OID。 否则 `_oid_` 是零。

如果 `INSERT` 命令包含 `RETURNING` 子句，其结果将和一个 `SELECT` 语句相同，包含那些基于该命令实际插入的行计算的、定义在 `RETURNING` 列表中的字段的值。

例子

向表 `films` 里插入单独一行：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在这个例子里面省略了字段 `len`，因此在它里面将存储缺省值：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在这个例子里，用 `DEFAULT` 子句作为日期字段，而不是声明一个数值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

插入一行完全由缺省值组成的数据行：

```
INSERT INTO films DEFAULT VALUES;
```

使用多行 `VALUES` 语法插入多行：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

从表 `tmp_films` 中插入几行到表 `films` 中， 字段布局与 `films` 相同：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

插入数组：

```
-- 创建一个空的3x3游戏板来玩圈-和-叉游戏(这些查询创建相同的板属性)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- 上述例子中的下标并非真正必须
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

向 `distributors` 表插入一个单独的行，并返回由 `DEFAULT` 子句生成的序列值：

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

增加销售人员为Acme Corporation的打折力度，并且在日志表中记录真个更新了的行和当前时间：

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

兼容性

`INSERT` 语句与 SQL 标准兼容。但 `RETURNING` 子句是 PostgreSQL扩展，就像在 `INSERT` 中使用 `WITH` 。 同样，省略字段名列表，但是并非所有字段都从 `VALUES` 子句或者 `_query_` 填充的这种用法是标准不允许的。

可能碰到的关于 `_query_` 子句特性的限制在 SELECT语句中有相关文档。

LISTEN

Name

LISTEN -- 监听一个通知

Synopsis

```
LISTEN _channel_
```

描述

`LISTEN` 将当前会话注册为通知通道 `_channel_` 的监听器。 如果当前会话已经被注册为该通知通道的监听器，那么什么也不做。

当执行了命令 `NOTIFY` `_channel_` 后，不管是此会话还是其它连接到同一数据库的会话，所有正在监听此通知通道的会话都将收到通知， 并且接下来每个会话将通知与其相连的前端应用。

使用 `UNLISTEN` 命令可以将一个会话内已注册的通知通道删除。同样， 会话退出时自动删除该会话正在监听的已注册通知通道。

前端应用检测通知事件的方法取决于PostgreSQL应用使用的编程接口。 如果使用基本的libpq库，那么应用将 `LISTEN` 当作普通 SQL 命令使用， 而且必须周期地调用 `PQnotifies` 过程来检测是否有通知到达。 其它像libpgtcl接口提供了更高级的控制通知事件的方法；实际上， 使用libpgtcl的应用程序员不应该直接使用 `LISTEN` 或 `UNLISTEN` 。 请参考你使用的接口的文档获取更多细节。

NOTIFY的手册页包含更广泛的关于 `LISTEN` 和 `NOTIFY` 的使用的讨论。

参数

`_channel_`

通知通道的名字，可以是任意标识符。

注意

`LISTEN` 在事务提交时生效。如果 `LISTEN` 或 `UNLISTEN` 在一个稍后回滚的事务中执行，那么被监听的通知通道的设置不会改变。

一个已经执行了 `LISTEN` 的事务不能准备两阶段提交。

例子

在psql里配制和执行一个监听/通知序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

兼容性

SQL 标准里没有 `LISTEN` 语句。

又见

NOTIFY, UNLISTEN

LOAD

Name

LOAD -- 加载一个共享库文件

Synopsis

```
LOAD '_filename_'
```

描述

这个命令加载一个共享库文件到PostgreSQL服务器的地址空间。 如果该文件已经被加载，那么这条命令什么也不做。包含C函数的共享库文件在其中之一的函数被调用时自动加载。 因此，一个明确的 `LOAD` 通常只需要加载一个修改服务器行为的库，通过"hooks" 而不是提供一组函数来修改服务器行为。

文件名是用和CREATE FUNCTION里描写的共享库的名字相同方法声明的； 特别要注意等是可以依赖搜索路径和自动附加系统标准共享库扩展名的特点。 参阅Section 35.9获取更多细节。

非超级用户仅可以将 `LOAD` 用于 `$libdir/plugins/` 中的库文件， 也就是说指定的 `_filename_` 必须精确的以该字符串开头。 数据库管理员有责任确保仅将"安全"的库文件安装在那里。

兼容性

`LOAD` 是PostgreSQL扩展。

又见

CREATE FUNCTION

LOCK

Name

LOCK -- 锁定一个表

Synopsis

```
LOCK [ TABLE ] [ ONLY ] _name_ [ * ] [, ...] [ IN _lockmode_ MODE ] [ NOWAIT ]

 这里的`_lockmode_`可以是下列之一：

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

描述

`LOCK TABLE` 获取一个表级锁，必要时等待任何冲突的锁释放。 如果声明了 `NOWAIT` ，那么 `LOCK TABLE` 并不等待它所需要的锁：如果无法立即获取该锁，那么该命令退出并且发出一个错误信息。 如果成功获取了这个锁，那么它就会在当前事务的余下部分一直保持。 没有 `UNLOCK TABLE` 命令；锁总是在事务结尾释放。

在为那些引用了表的命令自动请求锁的时候，PostgreSQL 总是尽可能使用最小限制的锁模式。 `LOCK TABLE` 是为你在需要更严格的锁的场合提供的。 例如，假设一个应用在"读已提交"隔离级别上运行事务，并且它需要保证在表中的数据在事务的运行过程中都存在。 要实现这个目的，你可以在查询之前对表使用 `SHARE` 锁模式进行锁定。 这样将保护数据不被并发修改并且为任何更进一步的对表的读操作提供实际的当前状态的数据， 因为 `SHARE` 锁模式与任何写操作需要的 `ROW EXCLUSIVE` 模式冲突， 并且你的 `LOCK TABLE` `_name_` IN SHARE MODE 语句将等到所有当前持有 `ROW EXCLUSIVE` 模式的锁提交或回卷后才执行。因此， 一旦你获得该锁，那么就不会存在未提交的写操作，并且其他人只能在你释放锁之后才能再次获取锁。

如果运行在 `可重复读` 或 `可串行化` 隔离级别实现类似的效果的时候， 你必须在执行任何 `SELECT` 或数据修改语句之前运行一个 `LOCK TABLE` 语句。 一个 `可重复读` 或 `可串行化` 事务的数据图像将在其第一个 `SELECT` 或者数据修改语句开始的时候冻结住。稍后的 `LOCK TABLE` 将仍然阻止并发的写， 但它不能保证事务读取的东西对应最近提交的数值。

如果一个此类的事务准备修改一个表中的数据，那么应该使用 `SHARE ROW EXCLUSIVE` 锁模式，而不是 `SHARE` 模式。这样就保证任意时刻只有一个此类的事务运行。不这样做就可能会死锁： 当两个并发的事务可能都请求 `SHARE` 模式，然后试图更改表中的数据时， 两个事务在实际执行更新的时候都需要 `ROW EXCLUSIVE` 锁模式，但是它们无法再次获取这个锁。 请注意，

一个事务自己的锁是从不冲突的，因此一个事务可以在持有 `SHARE` 模式的锁的时候请求 `ROW EXCLUSIVE` 模式(但是不能在任何其它事务持有 `SHARE` 模式的时候请求)。为了避免死锁，所有事务应该保证以相同的顺序对相同的对象请求锁， 并且，如果涉及多种锁模式，那么事务应该总是最先请求最严格的锁模式。

有关锁模式和锁定策略的更多信息，请参考Section 13.3。

参数

`_name_`

要锁定的现存表的名字(可以有模式修饰)。如果在表名前声明了 `ONLY`，那么只有那一个表被锁定。 如果没有声明 `ONLY`，那么该表和它的所有后代表（如果有）都被锁定。可选的，`*` 可以在表名后面指定，明确表明包含后代表。

命令 `LOCK TABLE a, b;` 等效于 `LOCK TABLE a; LOCK TABLE b;` 。 表是按照 `LOCK TABLE` 命令中声明的顺序一个接一个顺序上锁的。

`_lockmode_`

锁模式声明这个锁和哪些锁冲突。锁模式在Section 13.3里描述。

如果没有声明锁模式，那么使用最严格的模式 `ACCESS EXCLUSIVE` 模式。

`NOWAIT`

声明 `LOCK TABLE` 不去等待任何冲突的锁释放： 如果无法不等待立即获取所要求的锁，那么事务退出。

注意

`LOCK TABLE ... IN ACCESS SHARE MODE` 需要在目标表上有 `SELECT` 权限。 所有其它形式的 `LOCK` 需要表级别的 `UPDATE` ， `DELETE` 或 `TRUNCATE` 权限。

`LOCK TABLE` 在事务块的外部没什么用：锁依然被持有直到声明结束。 因此如果 `LOCK` 在一个事务块外面使用，PostgreSQL会报告一个错误。 使用BEGIN和COMMIT（或ROLLBACK) 定义一个事务块。

`LOCK TABLE` 只处理表级的锁，因此那些有 `ROW` 字样的锁都是用词不当。 这些模式名字通常应该理解为用户试图在一个被锁定的表中获取行级的锁。同样， `ROW EXCLUSIVE` 模式也是一个可共享的表级锁。一定要记住， 只要是涉及到 `LOCK TABLE`， 那么所有锁模式都有相同的语意， 区别只是它们与哪种锁冲突的规则。有关如何获取一个行级锁的信息，请参阅 Section 13.3.2和 `SELECT` 命令参考页的 锁定子句子句信息。

例子

演示在往一个外键表上插入时在有主键的表上使用 `SHARE` 的锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果记录没有返回则 ROLLBACK
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在执行删除操作时对一个有主键的表进行 `SHARE ROW EXCLUSIVE` 锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

在 SQL 标准里面没有 `LOCK TABLE`， 可以使用 `SET TRANSACTION` 来声明当前事务的级别。PostgreSQL也支持这个， 参阅SET TRANSACTION获取详细信息。

除了 `ACCESS SHARE`， `ACCESS EXCLUSIVE`， `SHARE UPDATE EXCLUSIVE` 锁模式外，PostgreSQL锁模式和 `LOCK TABLE` 语句都与那些在Oracle里面的兼容。

MOVE

Name

MOVE -- 定位一个游标

Synopsis

```
MOVE [ _direction_ [ FROM | IN ] ] _cursor_name_

where `_direction_` can be empty or one of:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE _count_
    RELATIVE _count_
    _count_
    ALL
    FORWARD
    FORWARD _count_
    FORWARD ALL
    BACKWARD
    BACKWARD _count_
    BACKWARD ALL
```

描述

`MOVE` 在不检索数据的情况下重新定位一个游标。 `MOVE` 的工作类似于 `FETCH` 命令，但只是重定位游标而不返回行。

`MOVE` 命令的参数与 `FETCH` 命令的参数相同； 请参考FETCH命令获取语法和参数的详细信息。

输出

成功完成时， `MOVE` 命令将返回一个下面形式的命令标签

```
MOVE _count_
```

`_count_` 是一个带同参数的 `FETCH` 命令会返回的行数(可能为零)。

例子

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

 -- 忽略开头5行:

MOVE FORWARD 5 IN liahona;
MOVE 5

-- Fetch the 6th row from the cursor liahona:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- 关闭游标liahona并提交事务:
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准里没有 `MOVE` 语句。

又见

CLOSE, DECLARE, FETCH

NOTIFY

Name

NOTIFY -- 生成一个通知

Synopsis

```
NOTIFY _channel_ [ , _payload_ ]
```

描述

`NOTIFY` 命令向当前数据库中所有执行过 `LISTEN` `_channel_`， 正在监听特定通道名字的前端应用发送一个通知事件和一个可选的"payload"字符串。 通知对所有用户可见。

`NOTIFY` 为访问同一个PostgreSQL 数据库的一组进程提供了一种简单的进程间通讯机制。负载字符串可以和通知一起发送， 并且传送结构化数据的更高级的机制可以通过使用数据库中的表从通知者传递数据到接收者。

传递给前端的通知事件包括通知通道名、发出通知的后端进程PID 和负载字符串，如果已经指定了字符串则该负载字符串为空。

定义将要用于给定数据库的通道名和每个意味着什么取决于数据库设计者。 通常，通知通道名与数据库里的表的名字相同，通知事件实际上意味着"我修改了此数据库， 请看一眼有什么新东西"。 `NOTIFY` 和 `LISTEN` 命令并不强制这种联系。例如，数据库设计者可以使用几个不同的通道名来标志一个表的几种不同改变。 另外，负载字符串可以用来区分各种情况。

当 `NOTIFY` 用于通知某一特定表修改的动作的发生， 一个实用的编程技巧是将 `NOTIFY` 放在一个由表更新触发的规则里。用这种方法， 通知将在表更新的时候自动触发，而且应用程序员不会碰巧忘记处理它。

`NOTIFY` 和 SQL 事务用某种重要的方法进行交换。首先，如果 `NOTIFY` 在事务内部执行，通知事件直到事务提交才会送出。这么做是有道理的，因为如果事务退出了， 那么在它里面的所有命令都没有效果(包括 `NOTIFY`)。但如果有人希望通知事件立即发送， 这就不太好了。其次，当一个正在监听的会话在一次事务内收到一个通知信号， 直到本次事务完成(提交或退出)之前，该通知事件将不被送到与之相连的客户端。同样， 如果一个通知在事务内部发送出去了，而该事务稍后又退出了，就希望通知可以在某种程度上被撤消， 因为通知一旦发送出去，服务器便不能从客户端"收回"通知，所以通知事件只是在事务之间传递。 这一点就要求使用 `NOTIFY` 作为实时信号的应用应该确保他们的事务尽可能短。

如果相同的通道名已经从相同的事务中发出了同样的负载字符串多次， 数据库服务器可以决定只送出一个字符串通知。另一方面，不同负载字符串的通知将总是作为不同通知传送。 相似的，来自不同事件的通知将永远不会被并入一个通知。除了删除稍后复制通知的实例，`NOTIFY`保证来自相同事务的通知以它们被传送的顺序到达。 也保证来自不同事务的信息以事务提交的顺序到达。

客户端经常会自己发送与正在监听的通知通道一样的`NOTIFY`。 这时它(客户端)也和其它正在监听的会话一样收到一个通知事件。 这样可能导致一些无用的工作(与应用逻辑有关)。例如， 对客户端刚写过的表又进行一次读操作以发现是否有更新。 可以通过检查服务器进程的PID(在通知事件中提供)是否与自己的会话的PID 一致(从libpq中取得)避免这样的额外工作。当他们一样时， 说明这是其自身回弹的信息，可以忽略。

参数

`_channel_`

生成信号(通知)的通知通道(任何标识符)。

`_payload_`

"payload"字符串与通知交流。必须指定为简单的字符串文本。缺省配置中必须少于8000字节。 （如果二进制数据或大量的信息需要交流，最好放在数据库表中并发送该记录的键字。）

注意

有一个已经发送的持有通知但目前还未被所有监听会话处理的序列。如果这个序列满了， 会话调用`NOTIFY`将会在提交时失败。序列是相当大的（标准安装中是8GB） 并且应该足够为几乎每个使用情况大。但是，如果会话执行`LISTEN`并且然后进入一个事务很长时间那么将不会有清除发生。一旦序列是半满的， 您将在日志中看到警告指向您的会话阻止清除。在这种情况下，应该确保这个会话结束他的当前事务， 这样清理能够进行。

一个已经执行了`NOTIFY`的事务不能准备两阶段提交。

pg_notify

也可以使用函数`pg_notify`(text , text) 发送一个通知。函数接受通道名作为第一个参数，负载作为第二个参数。 如果你需要
NOTIFY`命令更简单。

例子

在psql里配置和执行一个监听/通知序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from serv

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with
```

兼容性

SQL 标准里没有 `NOTIFY` 语句。

又见

LISTEN, UNLISTEN

PREPARE

Name

PREPARE -- 创建一个预备语句

Synopsis

```
PREPARE _name_ [ ( _data_type_ [, ...] ) ] AS _statement_
```

描述

`PREPARE` 创建一个预备语句。一个预备语句是服务器端的对象， 可以用于优化性能。在执行 `PREPARE` 语句的时候，指定的查询被解析、分析、 重写。当随后发出 `EXECUTE` 语句的时候，预备语句被规划和执行。 这种分工避免了重复解析分析工作，允许执行计划依赖于所提供的特定的参数值。

预备语句可以接受参数：在它执行的时候替换到查询中的数值。可以在一个预备语句里按照位置来引用参数， 比如 `$1` ， `$2` 等。可以指定一个相应的参数数据类型列表。 如果一个参数的数据类型没有被指定或声明为 `unknown` ， 那么其类型将根据该参数所使用的实际上下文环境进行推测(如果有可能的话)。当执行该语句的时候， 将在 `EXECUTE` 语句中为这些参数指定实际值。参见EXECUTE获取更多信息。

预备语句只是在当前数据库会话的过程中存在。如果客户端退出，那么预备语句就会被遗忘， 因此必须在被重新使用之前重新创建。这也意味着一个预备语句不能被多个数据库客户端同时使用； 但是，每个客户端可以创建它们自己的预备语句来使用。预备语句可以用DEALLOCATE命令手工清除。

如果一个会话准备用于执行大量类似的查询，那么预备语句可以获得最大限度的性能优势。 如果查询非常复杂，需要复杂的规划或者重写，那么性能差距将更加明显。比如， 如果查询设计许多表的连接，或者有多种规则要求应用。如果查询的规划和重写相对简单， 而执行起来开销相当大，那么预备语句的性能优势就不那么明显。

参数

`_name_`

给予这个特定的预备语句任意名字。它必须在一个会话中是唯一的， 并且用于执行或者删除一个预备语句。

`_data_type_`

预备语句的某个参数的数据类型。如果某个参数的数据类型未指定或指定为 `unknown` ， 那么将根据该参数使用的上下文环境进行推断。可以使用 `$1` , `$2` 等等在预备语句内部引用这个参数。

`_statement_`

`SELECT` , `INSERT` , `UPDATE` , `DELETE` , 或 `VALUES` 语句之一。

注意

如果预备语句执行了足够长的时间，服务器可能最终决定保存并重新使用一个一般的计划，而不是每次重新计划。如果预备语句没有参数，那么将立即发生； 否则只在一般计划看起来不比依赖于特定参数值的计划昂贵时发生。 典型的，只有查询的性能对提供的特定的参数值相当迟钝时选择一般的计划。

为了查询PostgreSQL用于预备语句的查询计划，使用EXPLAIN。 如果正在使用一个一般的计划，它将包含参数符号 `$``_n_` ， 此时定制计划将有代入的当前实际参数的值。

有关查询规划和PostgreSQL为查询优化的目的收集统计的更多信息， 参阅ANALYZE文档。

尽管预备语句的重点是避免重复分析和规划语句，但PostgreSQL 在使用它之前强制重复分析和重复计划语句， 每当在语句中使用的数据库对象自前一次使用预备语句后经历了明确(DDL)的变化时。 还有，如果search_path的值在到下一个使用时改变了， 那么语句将使用新的 `search_path` 重新分析。 （后面的行为是 PostgreSQL 9.3新增的。） 使用预备语句语义的规则几乎等于反复重新提交相同的查询文本， 但是如果没有对象定义改变那么将有一个性能优势，尤其是保持使用最佳的规划。 语义等价不完美的一个例子是如果语句引用一个未经限定名字的表， 然后在一个模式中创建了一个相同名字的表出现在 `search_path` 中靠前的地方， 那么将不会自动重复分析，因为用于语句的对象没有改变。但是，如果其他改变强制了一个重复分析， 那么新表将在后来的使用中引用。

可以通过查询 `pg_prepared_statements` 系统视图获得某个会话中所有可用的预备语句。

Examples

为一个 `INSERT` 语句创建一个预备语句然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个 `SELECT` 语句创建一个预备语句然后执行它：

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

注意，第二个参数的数据类型并未指定。所以将从上下文环境推测 `$2` 的类型。

兼容性

SQL 标准包含一个 `PREPARE` 语句，但是它只用于嵌入式 SQL 。 PostgreSQL 实现的 `PREPARE` 语句的语法也略有不同。

又见

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

Name

PREPARE TRANSACTION -- 为当前事务做两阶段提交的准备

Synopsis

```
PREPARE TRANSACTION _transaction_id_
```

描述

`PREPARE TRANSACTION` 为当前事务的两阶段提交做准备。在命令之后， 事务就不再和当前会话关联了；它的状态完全保存在磁盘上，它被提交成功的可能性非常高， 即使是在请求提交之前数据库发生了崩溃也如此。

一旦准备好了，一个事务就可以在稍后用COMMIT PREPARED或 ROLLBACK PREPARED命令分别进行提交或者回滚。 这些命令可以从任何会话中发出，而不光是最初执行事务的那个会话。

从发出命令的会话的角度来看， `PREPARE TRANSACTION` 不同于 `ROLLBACK` ： 在执行它之后，就不再有活跃的当前事务了，并且预备事务的效果无法见到 (在事务提交的时候其效果会再次可见)。

如果 `PREPARE TRANSACTION` 因为某些原因失败， 那么它就会变成一个 `ROLLBACK` ，当前事务被取消。

参数

`_transaction_id_`

一个任意的标识符，用于后面在 `COMMIT PREPARED` 或 `ROLLBACK PREPARED` 的时候标识这个事务。这个标识符必须以字符串文本的方式书写，并且必须小于 200 字节长。 它不能和任何当前预备事务已经使用了的标识符同名。

注意

`PREPARE TRANSACTION` 不是为了在应用或交互会话中使用。 它的目的是允许外部的事务管理器实现多个数据库或者与其他事务源实现原子的全局事务。 除非你写一个事务管理器，否则你不应该使用 `PREPARE TRANSACTION` 。

这条命令必须在一个事务块里面使用。用BEGIN开始一个事务。

目前，不允许对那些执行了涉及临时表，或者是创建了带 `WITH HOLD` 游标， 或者执行了 `LISTEN` 或 `UNLISTEN` 的事务进行 `PREPARE` 。 这些特性和当前会话绑定得实在是太紧密了，因此在一个预备事务里没什么可用的。

如果事务用 `SET` （没有 `LOCAL` 选项）修改了运行时参数， 这些效果在 `PREPARE TRANSACTION` 之后保留，并且不会被任何以后的 `COMMIT PREPARED` 或 `ROLLBACK PREPARED` 所影响。 因此，在这方面， `PREPARE TRANSACTION` 表现得更像 `COMMIT` 而不是 `ROLLBACK` 。

所有目前可用的预备事务都在系统视图 `pg_prepared_xacts` 里面列出。

> **Caution**
>
> 把一个事务长时间停在预备状态是不明智的：它会影响 `VACUUM` 回收存储的能力， 并且在极端情况可能导致数据库关闭以阻止包括事务ID（参阅Section 23.1.5）。 还要记住，事务会继续持有它们持有的锁。 这个特性的用法是预备事务通常会在外部的事务管理器核实了其它数据库也准备好提交之后，提交或者回滚事务。

如果你还没有设置一个额外的事务管理器追踪预备事务和确保他们得到及时关闭， 最好通过设置max_prepared_transactions为0来保持预备事务特征不可用。 这将阻止意外的创建可能被遗忘和最终导致问题的预备事务。 |

例子

为当前事务做两阶段提交的准备，使用 `foobar` 做为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

兼容性

`PREPARE TRANSACTION` 是一个PostgreSQL扩展。 该命令设计是给外部事务管理系统使用，SQL标准只涉及部分功能（如X/Open XA），但这些功能目前没有标准化。

又见

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

Name

REASSIGN OWNED -- 修改数据库对象的属主

Synopsis

```
REASSIGN OWNED BY _old_role_ [, ...] TO _new_role_
```

描述

`REASSIGN OWNED` 要求系统将所有 old_roles 拥有的数据库对象的属主更改为 new_role。

参数

`_old_role_`

旧属主的角色名。当前数据库和所有共享对象(数据库，表空间) 中该角色所拥有的所有对象的属主将改为 `_new_role_` 。

`_new_role_`

将要成为这些对象属主的新角色的名字

注意

`REASSIGN OWNED` 常用于在删除角色之前的准备工作。因为 `REASSIGN OWNED` 不影响其他数据库中的对象，所以必须在即将删除的角色拥有对象的每一个数据库中执行该命令。

`REASSIGN OWNED` 要求原角色和目标角色上的权限。

DROP OWNED可以用来删除角色所拥有的所有对象。 还请注意 `DROP OWNED` 只需要原角色的权限。

`REASSIGN OWNED` 并不影响 old_roles 在不被其拥有的对象上的权限。 使用 `DROP OWNED` 来删除这些权限。

兼容性

`REASSIGN OWNED` 语句是一个PostgreSQL扩展。

又见

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

Name

REFRESH MATERIALIZED VIEW -- 替换物化视图的内容

Synopsis

```
REFRESH MATERIALIZED VIEW _name_
    [ WITH [ NO ] DATA ]
```

描述

`REFRESH MATERIALIZED VIEW` 完全替换一个物化视图的内容。旧的内容被丢弃。 如果声明了 `WITH DATA` （或缺省）， 后端查询被执行以提供新的数据， 物化视图留在可扫描的状态。 如果声明了 `WITH NO DATA` ， 那么不会产生新的数据，并且物化视图留在一个不可扫描的状态。

参数

`_name_`

要刷新内容的物化视图的名字（可以有模式修饰）。

注意

当保留了未来的CLUSTER操作的缺省索引时， `REFRESH MATERIALIZED VIEW` 不要求生成的行基于这个属性。如果你希望数据根据生成排序，必须在后端查询中使用一个 `ORDER BY` 子句。

例子

这个命令将刷新物化视图 `order_summary` 的内容，使用来自物化视图定义的查询， 并且使它留在可扫描的状态：

```
REFRESH MATERIALIZED VIEW order_summary;
```

这个命令将清空物化视图 `annual_statistics_basis` 的存储区，并使它留在不可扫描的状态：

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

兼容性

`REFRESH MATERIALIZED VIEW` 是一个PostgreSQL扩展。

又见

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

Name

REINDEX -- 重建索引

Synopsis

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } _name_ [ FORCE ]
```

描述

`REINDEX` 使用存储在索引表中的数据重建索引，替换旧的索引的副本。 使用 `REINDEX` 有两个主要原因：

- 索引崩溃，并且不再包含有效的数据。尽管理论上这是不可能发生的， 但实际上索引会因为软件毛病或者硬件问题而崩溃。 `REINDEX` 提供了一个恢复方法。
- 索引变得"臃肿"，包含大量的空页或接近空页。 这个问题在某些罕见访问模式时会发生在 B-tree 索引上。 `REINDEX` 通过写一个不带无用索引页的新索引提供了缩小索引空间消耗的途径。 参阅Section 23.2获取更多信息。
- 为索引更改了存储参数(例如填充因子)，并且希望这个更改完全生效。
- 使用 `CONCURRENTLY` 选项创建索引失败，留下了一个"非法"索引。 这样的索引毫无用处，但是可以通过 `REINDEX` 重建新索引来覆盖。注意， `REINDEX` 不能并发创建。要在生产环境中重建索引并且尽可能减小对尖峰负载的影响， 可以先删除旧索引，然后使用 `CREATE INDEX CONCURRENTLY` 命令重建新索引。

参数

`INDEX`

重新建立指定的索引。

`TABLE`

重新建立指定表的所有索引。如果表有从属的"TOAST"表，那么这个表也会重新索引。

`DATABASE`

重建当前数据库里的所有索引。也处理在共享系统表上的索引。 这种形式的 `REINDEX` 不能在事务块中执行。

`SYSTEM`

在当前数据库上重建所有系统表上的索引。包括在共享系统表上的索引。不会处理在用户表上的索引。 这种形式的 `REINDEX` 不能在事务块中执行。

`_name_`

需要重建索引的索引、表、数据库的名称。表和索引名可以有模式修饰。目前, `REINDEX DATABASE` 和 `REINDEX SYSTEM` 只能重建当前数据库的索引， 因此其参数必须匹配当前数据库的名字。

`FORCE`

这是一个废弃的选项，如果声明，会被忽略。

注意

如果你怀疑一个用户表上的索引崩溃了，你可以简单地使用 `REINDEX INDEX` 重建该索引，或者使用 `REINDEX TABLE` 重建该表上的所有索引。

如果你从一个崩溃的系统表索引上恢复，事情会更棘手一些。这种情况下， 系统必须不能使用任何有疑问的索引。实际上，在这种情况下， 你可能发现服务器进程在启动之后马上就崩溃了，因为依赖于崩溃了的索引。要想安全恢复， 服务器必须带着 `-P` 选项启动，它禁止服务器在查找系统表的时候使用索引。

达到这个目的的一个方法是停止服务器然后带着 `-P` 命令行选项启动一个独立的PostgreSQL服务器。然后， 根据你希望恢复的程度，发出 `REINDEX DATABASE`，`REINDEX SYSTEM`，`REINDEX TABLE`，`REINDEX INDEX` 命令。如果还有怀疑， 使用 `REINDEX SYSTEM` 选择重新构造数据库中全部的系统索引。 然后退出独立服务器会话并且重启普通的服务器。参阅postgres 手册页获取有关如何与独立服务器交互的信息。

另外，一个普通的会话可以在其命令行选项里使用 `-P` 选项启动。 这么做的方法因不同的客户端而异，但是在所有基于libpq的客户端上， 都可以通过在启动客户端之前设置 `PGOPTIONS` 环境变量为 `-P` 来实现。 请注意尽管这个方法并不要求锁住其它客户端，但是禁止其它客户端连接受损的数据库， 直到完成修补是一个明智的选择。

`REINDEX` 类似于删除并重建索引，表现在它们都是从零开始重建。不过， 从锁的角度考虑，两者是有区别的。 `REINDEX` 锁住对索引的父表的写操作， 但是不锁读操作。并且它还在被处理的特定索引上保持一个排他锁， 这样它将阻止试图使用该索引的读操作。相比之下， `DROP INDEX` 在父表上短暂的保持一个排他锁，同时锁住读和写。随后的 `CREATE INDEX` 锁住写操作但是不会锁住读操作；因为索引还不存在，所以不会有试图使用它的读操作， 意味着操作中不会有阻塞，只不过读操作会被迫只能使用顺序扫描。

对一个索引或者表进行重建索引，要求你是该索引或者表的所有者。 对一个数据库重建索引要求你是该数据库的所有者(注意，可以重建其它用户拥有的索引)。 当然，超级用户总是可以重建所有索引。

PostgreSQL 8.1之前， `REINDEX DATABASE` 只处理系统索引，而不是人们从名字猜测的那样，处理所有索引。这个行为现在已经改变了， 以减少意外的因素。旧的行为可以通过 `REINDEX SYSTEM` 获得。

PostgreSQL 7.4之前， `REINDEX TABLE` 并不自动处理 TOAST 表，因此这些表必须用独立的命令进行处理。这么做仍然可以， 但是已经多余了。

例子

重建一个单独的索引：

```
REINDEX INDEX my_index;
```

重建表 `my_table` 上的所有索引：

```
REINDEX TABLE my_table;
```

重建一个数据库上的所有系统索引，不管系统索引是否仍然有效：

```
$ <kbd class="literal">export PGOPTIONS="-P"</kbd>
$ <kbd class="literal">psql broken_db</kbd>
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

兼容性

SQL 标准里没有 `REINDEX` 命令。

RELEASE SAVEPOINT

Name

RELEASE SAVEPOINT -- 删除一个先前定义的保存点

Synopsis

```
RELEASE [ SAVEPOINT ] _savepoint_name_
```

描述

`RELEASE SAVEPOINT` 删除一个当前事务先前定义的保存点。

把一个保存点删除就令其无法作为回滚点使用，除此之外它没有其它用户可见的行为。 它并不能撤销在保存点建立起来之后执行的命令的影响。要撤销那些命令可以使用ROLLBACK TO SAVEPOINT。 在不再需要的时候删除一个保存点可以令系统在事务结束之前提前回收一些资源。

`RELEASE SAVEPOINT` 也删除指定保存点之后建立的所有保存点。

参数

`_savepoint_name_`

要删除的保存点的名字。

注意

声明一个以前没有定义的保存点名字将导致错误。

如果事务在回滚状态，则不能释放保存点。

如果多个保存点拥有同样的名字，只有最近定义的那个才被释放。

例子

建立并稍后删除一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将同时插入 3 和 4 。

兼容性

这条命令遵循SQL标准。标准规定关键字 `SAVEPOINT` 是必须的。 但PostgreSQL允许省略它。

又见

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

Name

RESET -- 把一个运行时参数重置为缺省值

Synopsis

```
RESET _configuration_parameter_
RESET ALL
```

描述

`RESET` 将运行时参数恢复为缺省值。 `RESET` 是下面语句的一个变种

```
SET _configuration_parameter_ TO DEFAULT
```

请参考SET命令获取详细信息。

缺省值是变量可以拥有并且在当前会话中没有用 `SET` 设置过的值。 这个数值的实际源头可能是编译时的缺省：配置文件、命令行开关、针对每个数据库或每用户的缺省设置。 这与将它定义为"在会话开始时该参数的值"有轻微的不同，因为如果值来自配置文件， 将会被重新设置为现在配置文件中声明的东西。参阅Chapter 18获取细节。

`RESET` 的事务性行为和 `SET` 相同：它的影响将会被事务回滚撤销。

参数

`_configuration_parameter_`

可预置的运行时参数名。可用的参数在Chapter 18和SET 参考页中记录。

`ALL`

把所有运行时参数设置为缺省值。

例子

把 `timezone` 配置变量设为其缺省值：

```
RESET timezone;
```

兼容性

`RESET` 是PostgreSQL扩展。

又见

SET, SHOW

REVOKE

Name

REVOKE -- 删除访问权限

Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] _table_name_ [, ...]
         | ALL TABLES IN SCHEMA _schema_name_ [, ...] }
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( _column_name_ [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( _column_name_ [, ...] ) }
    ON [ TABLE ] _table_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE _sequence_name_ [, ...]
         | ALL SEQUENCES IN SCHEMA _schema_name_ [, ...] }
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE _database_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN _domain_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER _fdw_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER _server_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION _function_name_ ( [ [ _argmode_ ] [ _arg_name_ ] _arg_type_ [, ...] ] )
         | ALL FUNCTIONS IN SCHEMA _schema_name_ [, ...] }
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
```

```
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE _lang_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT _loid_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA _schema_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE _tablespace_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE _type_name_ [, ...]
    FROM { [ GROUP ] _role_name_ | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    _role_name_ [, ...] FROM _role_name_ [, ...]
    [ CASCADE | RESTRICT ]
```

描述

`REVOKE` 撤销以前赋予一个或多个角色的权限。关键字 `PUBLIC` 代表隐含定义的、拥有所有角色的组。

参阅GRANT命令的描述获取权限类型的含义。

请注意，任何特定的角色都将拥有直接赋予它的权限，加上它所在组的权限， 再加上赋予 `PUBLIC` 的权限的总和。因此，举例来说，废止 `PUBLIC` 的 `SELECT` 权限并不意味着所有角色都失去了对该对象的 `SELECT` 权限： 那些直接得到的权限以及通过一个组得到的权限仍然有效。相似的，废止一个用户的 `SELECT` 权限可能并不阻止用户使用 `SELECT` ，如果 `PUBLIC` 或其他成员组仍然拥有 `SELECT` 权限。

如果指定了 `GRANT OPTION FOR` ，那么只是撤销对该权限的授权的权力， 而不是撤销该权限本身。否则，权限和授权选项都被撤销。

如果一个用户持有某个权限，并且还有授权的选项，并且还把这个权限赋予了其它用户， 那么那些其它用户持有的权限都叫做依赖性权限。如果第一个用户持有的权限或者授权选项被撤销， 而依赖性权限仍然存在；那么如果声明了 `CASCADE` ，则所有依赖性权限都被撤销， 否

则撤销动作就会失败。这个递规的撤销只影响那种通过一个用户链赋予的权限， 这个链条可以通过这条 `REVOKE` 命令里面给出的用户跟踪。因此， 如果权限本身是通过其它用户赋予的，那么被影响的用户可以有效地保留这个权限。

当在表上撤销权限时，相应的字段权限（如果有）也自动撤销。另一方面， 如果已经赋予了一个角色在表上的权限，那么在单独的字段上删除相同的权限将不会有作用。

在撤销一个角色里的成员关系的时候，不是调用 `ADMIN OPTION` 而是调用 `GRANT OPTION` ，但是行为类似。不过这种形式的命令不允许出现 `GROUP` 噪声字。

注意

使用psql的 `\dp` 命令显示在一个现存表和字段上赋予的权限。 又见GRANT获取关于格式的信息。对于非表对象，可以使用 `\d` 命令显示他们的权限。

一个用户只能撤销由它自己直接赋予的权限。举例来说，如果用户 A 带着授权选项把一个权限赋予了用户 B， 然后用户 B 又赋予了用户 C，那么用户 A 不能直接将 C 的权限撤销。但是， 用户 A 可以撤销用户 B 的授权选项，并且使用 `CASCADE` 选项，这样， 用户 C 的权限就会自动被撤销。另外一个例子：如果 A 和 B 都赋予了 C 同样的权限， 则 A 可以撤销他自己的授权选项，但是不能撤销 B 的，因此 C 仍然有效地拥有该权限。

如果一个对象的非所有者试图 `REVOKE` 对象上的权限，那么，如果这个用户没有该对象上的权限， 则命令马上失败。只要他有某些权限，则命令继续，但是它只撤销那些该用户有授权选项的权限。 如果没有在授权选项，那么 `REVOKE ALL PRIVILEGES` 形式将发出一个错误信息， 而对于其它形式的命令而言，如果同样是命令中指定名字的权限没有相应的授权选项， 那么该命令将发出一个警告。原则上这些语句也适用于对象所有者， 但是因为所有者总是认为持有所有授权选项，所以这种情况绝不会发生。

如果一个超级用户发出一个 `GRANT` 或 `REVOKE` 命令， 那么命令是以被影响的对象的所有者执行的。因为所有权限最终从对象所有者 (可能间接通过赋权选项)获取，超级用户可以废除所有权限， 但是这样就要求像上面说的那样使用 `CASCADE` 。

`REVOKE` 也可以由一个并非被影响对象的所有者来执行， 不过这个角色必须是拥有该对象的角色的成员，或者是一个在该对象上持有 `WITH GRANT OPTION` 的角色的成员。在这种情况下， 该命令执行起来就好像是由实际拥有该对象的角色，或者是在该对象上持有 `WITH GRANT OPTION` 权限的角色发出的一样。比如，如果表 `t1` 被 `g1` 所有，而 `u1` 是 `g1` 的成员，那么 `u1` 可以撤销 `t1` 上的权限，而纪录为 `g1` 发出的命令。这种现像包括 `u1` 和其它 `g1` 角色成员发出的授权。

如果执行 `REVOKE` 的角色持有权限是通过多层成员关系获得的， 那么具体是哪个包含的角色执行的该命令就是未声明的。在这种场合下， 最好的方法是使用 `SET ROLE` 成为你希望执行 `REVOKE` 的角色。 不这么做的后果可能导致删除你不想删除的权限，或者是啥权限都没有删除。

例子

撤销公众在表 `films` 上的插入权限：

```
REVOKE INSERT ON films FROM PUBLIC;
```

撤销用户 `manuel` 对视图 `kinds` 的所有权限：

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

请注意这样实际上意味着"撤销所有我赋予的权限"。

删除用户 `joe` 的 `admins` 成员关系：

```
REVOKE admins FROM joe;
```

兼容性

GRANT命令的兼容性信息基本上也适用于 `REVOKE` 。 标准要求 `RESTRICT` 或 `CASCADE` 必须出现， 但是PostgreSQL假设缺省是 `RESTRICT` 。

又见

GRANT

ROLLBACK

Name

ROLLBACK -- 退出当前事务

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

描述

`ROLLBACK` 回滚当前事务并取消当前事务中的所有更新。

参数

`WORK``TRANSACTION`

可选的关键字。没有作用。

注意

使用COMMIT语句将对事务进行提交。

如果不在一个事务内部发出 `ROLLBACK` 不会有问题，但是将抛出一个警告信息。

例子

取消所有更改：

```
ROLLBACK;
```

兼容性

SQL 标准只声明了两种形式 `ROLLBACK` 和 `ROLLBACK WORK` 。 否则完全兼容。

又见

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

Name

ROLLBACK PREPARED -- 取消一个先前为两阶段提交准备好的事务

Synopsis

```
ROLLBACK PREPARED _transaction_id_
```

描述

`ROLLBACK PREPARED` 回滚一个处于准备好状态的事务。

参数

`_transaction_id_`

要回滚掉的事务的事务标识符。

注意

要想回滚掉一个预备事务，你必须要么是最初发起事务的用户，要么是超级用户。 但你不必在执行事务的同一个会话里。

这条命令不能在事务块里执行。预备事务将被马上回滚。

所有当前可用的预备事务都在系统视图 `pg_prepared_xacts` 中列出。

例子

回滚一个使用事务标识符 `foobar` 标识的事务：

```
ROLLBACK PREPARED 'foobar';
```

兼容性

`ROLLBACK PREPARED` 是PostgreSQL的扩展。 是为外部事务管理系统设计的，一些被标准(比如X/Open XA)所覆盖， 但是SQL方面的这些系统是非标准化的。

又见

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

Name

ROLLBACK TO SAVEPOINT -- 回滚到一个保存点

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] _savepoint_name_
```

描述

回滚所有指定保存点建立之后执行的命令。保存点仍然有效，并且需要时可以再次回滚到该点。

`ROLLBACK TO SAVEPOINT` 隐含地删除所有在该保存点之后建立的保存点。

参数

`_savepoint_name_`

回滚截至的保存点。

注意

使用RELEASE SAVEPOINT删除一个保存点， 而不会抛弃这个保存点建立之后执行的命令结果。

声明一个还没有建立的保存点名字是一个错误。

在保存点方面，游标有一些非事务性的行为。任何在保存点里打开的游标都会在回滚掉这个保存点之后关闭。 如果一个前面打开了的游标在保存点里面，并且游标被一个 `FETCH` 或 `MOVE` 命令影响， 而这个保存点稍后回滚了，那么这个游标仍然在 `FETCH` 让它指向的位置 (也就是，由 `FETCH` 造成的游标动作不会被回滚)。关闭一个游标的行为也不会被回滚给撤消掉。 但是，如果由游标查询引起的副作用（如查询调用不稳定函数引起的副作用）在一个稍后会回滚的保存点发生， 那么他们也会回滚。如果一个游标的操作导致事务回滚，那么这个游标就会置于不可执行状态， 所以，尽管一个事务可以用 `ROLLBACK TO SAVEPOINT` 重新恢复，但是游标不能再使用了。

例子

撤销 `my_savepoint` 建立之后执行的命令的影响：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不受保存点回滚的影响：

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

兼容性

SQL标准声明关键字 `SAVEPOINT` 是必须的，但是PostgreSQL 和Oracle允许省略 `SAVEPOINT` 关键字。SQL 只允许 `WORK` 而不是 `TRANSACTION` 作为 `ROLLBACK` 后面的无意义关键字。还有，SQL 有一个可选的 `AND [ NO ] CHAIN` 子句，目前PostgreSQL还不支持。 否则，这个命令完全兼容 SQL 标准。

又见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

Name

SAVEPOINT -- 在当前事务里定义一个新保存点

Synopsis

```
SAVEPOINT _savepoint_name_
```

描述

`SAVEPOINT` 在当前事务里建立一个新的保存点。

保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚， 把事务的状态恢复到保存点所在的时刻。

参数

`_savepoint_name_`

赋予新保存点的名字。

注意

使用ROLLBACK TO SAVEPOINT回滚到一个保存点。使用RELEASE SAVEPOINT 删除一个保存点，但是保留该保存点建立后执行的命令的效果。

保存点只能在一个事务块里面建立。在一个事务里面可以定义多个保存点。

例子

建立一个保存点，稍后撤销这个保存点建立后执行的所有命令的结果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入数值 1 和 3， 而不会插入 2 。

建立并稍后删除一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4 。

兼容性

SQL 要求在另外一个同名保存点建立的时候自动删除前面那个同名保存点。 在PostgreSQL 里， 将保留旧的保存点， 但是在回滚或者释放的时候， 只使用最近的那个。 用 `RELEASE SAVEPOINT` 释放了新的保存点将导致旧的再次成为 `ROLLBACK TO SAVEPOINT` 和 `RELEASE SAVEPOINT` 可以访问的保存点。 否则， `SAVEPOINT` 是完全符合 SQL 标准的。

又见

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

Name

SECURITY LABEL -- 定义或改变一个应用于对象的安全标签

Synopsis

```
SECURITY LABEL [ FOR _provider_ ] ON
{
  TABLE _object_name_ |
  COLUMN _table_name_._column_name_ |
  AGGREGATE _agg_name_ (_agg_type_ [, ...] ) |
  DATABASE _object_name_ |
  DOMAIN _object_name_ |
  EVENT TRIGGER _object_name_ |
  FOREIGN TABLE _object_name_
  FUNCTION _function_name_ ( [ [ _argmode_ ] [ _argname_ ] _argtype_ [, ...] ] ) |
  LARGE OBJECT _large_object_oid_ |
  MATERIALIZED VIEW _object_name_ |
  [ PROCEDURAL ] LANGUAGE _object_name_ |
  ROLE _object_name_ |
  SCHEMA _object_name_ |
  SEQUENCE _object_name_ |
  TABLESPACE _object_name_ |
  TYPE _object_name_ |
  VIEW _object_name_
} IS '_label_'
```

描述

`SECURITY LABEL` 为一个数据库对象申请一个安全标签。 每一个标签提供程序的任意数量的安全标签都可以与一个给定的数据库对象关联。 标签提供者是可加载模块，通过使用函数 `register_label_provider` 记录他们自己。

> **Note:** `register_label_provider` 不是SQL函数；只能从C代码存入后端调用。

标签提供者决定一个给定的标签是否有效，并且是否允许将那个标签分配给一个给定的对象。 给定标签的含义和标签提供者的描述相同。PostgreSQL 不限制标签提供者是否或如何解释安全标签；只是提供过一个存储它们的机制。实际上， 这个便利是为了允许集成基于标签的强制访问控制（MAC）系统，如SE-Linux。 这样的系统使得访问控制决策基于对象标签，而不是传统的自主访问控制（DAC）概念，如用户和组。

参数

`_object_name_``_table_name.column_name_` `_agg_name_` `_function_name_`

有标签的对象的名字。可模式修饰的表、集群、域、外部表、函数、序列、类型和视图的名字。

`_provider_`

与这个标签相关的提供者的名字。被指名的提供者必须被加载并且必须同意提出的标签操作。 如果只加载了一个提供者，那么为了简洁会省略提供者的名字。

`_arg_type_`

聚集函数操作的输入数据类型。要引用一个零参数的聚集函数，在输入数据类型的列表位置写 `*` 。

`_argmode_`

函数参数的模式： `IN` , `OUT` , `INOUT` , 或 `VARIADIC` 。 如果省略，缺省是 `IN` 。请注意， `SECURITY LABEL ON FUNCTION` 并不实际注意 `OUT` 参数，因为只需要输入参数判断函数的安全。 所以列出 `IN` , `INOUT` , 和 `VARIADIC` 就足够了。

`_argname_`

函数参数的名字。请注意， `SECURITY LABEL ON FUNCTION` 并不实际注意参数名字，因为只需要参数数据类型判断函数的身份。

`_argtype_`

如果有，是函数参数的数据类型（可以有模式修饰）。

`_large_object_oid_`

大对象的OID。

`PROCEDURAL`

这是一个噪声字。

`_label_`

新的安全标签，写作一个字符串；或 `NULL` 以删除安全标签。

例子

下列例子显示了如何改变一个表的安全标签。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

兼容性

SQL标准中没有`SECURITY LABEL`命令。

又见

sepgsql, dummy_seclabel

SELECT

Name

SELECT, TABLE, WITH -- 从表或视图中取出若干行

Synopsis

```
[ WITH [ RECURSIVE ] _with_query_ [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( _expression_ [, ...] ) ] ]
    * | _expression_ [ [ AS ] _output_name_ ] [, ...]
    [ FROM _from_item_ [, ...] ]
    [ WHERE _condition_ ]
    [ GROUP BY _expression_ [, ...] ]
    [ HAVING _condition_ [, ...] ]
    [ WINDOW _window_name_ AS ( _window_definition_ ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] _select_ ]
    [ ORDER BY _expression_ [ ASC | DESC | USING _operator_ ] [ NULLS { FIRST | LAST } ]
    [ LIMIT { _count_ | ALL } ]
    [ OFFSET _start_ [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ _count_ ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF _table_name_ [, ...] ] [ NO

这里的`_from_item_` 可以是:

    [ ONLY ] _table_name_ [ * ] [ [ AS ] _alias_ [ ( _column_alias_ [, ...] ) ] ]
    [ LATERAL ] ( _select_ ) [ AS ] _alias_ [ ( _column_alias_ [, ...] ) ]
    _with_query_name_ [ [ AS ] _alias_ [ ( _column_alias_ [, ...] ) ] ]
    [ LATERAL ] _function_name_ ( [ _argument_ [, ...] ] ) [ AS ] _alias_ [ ( _column_ali
    [ LATERAL ] _function_name_ ( [ _argument_ [, ...] ] ) AS ( _column_definition_ [, ..
    _from_item_ [ NATURAL ] _join_type_ _from_item_ [ ON _join_condition_ | USING ( _join

`_with_query_`是:

    _with_query_name_ [ ( _column_name_ [, ...] ) ] AS ( _select_ | _values_ | _insert_ |

TABLE [ ONLY ] _table_name_ [ * ]
```

描述

`SELECT` 将从零个或更多表中返回记录行。 `SELECT` 通常的处理如下：

1. 计算列出在 `WITH` 中的所有查询。这些有效的充当可以在 `FROM` 列表中引用的临时表。一个在 `FROM` 中多次引用的 `WITH` 查询只计算一次。 （参阅下面的 *WITH 子句*。）

2. 计算列出在 `FROM` 中的所有元素(`FROM` 列表中的每个元素都是一个实际的或虚拟的表)。如果在 `FROM` 列表里声明了多个元素，那么他们就交叉连接在一起(参见下面的 *FROM 子句*)。

3. 如果声明了 `WHERE` 子句，那么在输出中消除所有不满足条件的行。 （参见下面的*WHERE 子句*。）

4. 如果声明了 `GROUP BY` 子句，输出就组合成匹配一个或多个数值的不同组里。 如果出现了 `HAVING` 子句，那么它消除那些不满足给出条件的组。 （参见下面的*GROUP BY 子句*和 *HAVING 子句*。）

5. 实际输出行将使用 `SELECT` 输出表达式针对每一个选中的行或行组进行计算。 （参见下面的*SELECT 列表*。）

6. `SELECT DISTINCT` 从结果中消除重复的行。 `SELECT DISTINCT ON` 消除匹配所有指定表达式的行。 `SELECT ALL` （缺省）返回所有的行， 包括重复的行。（参阅下面的*DISTINCT 子句*。）

7. 使用 `UNION` ， `INTERSECT` ， `EXCEPT` 可以把多个 `SELECT` 语句的输出合并成一个结果集。 `UNION` 操作符返回两个结果集的并集。 `INTERSECT` 操作符返回两个结果集的交集。 `EXCEPT` 操作符返回在第一个结果集对第二个结果集的差集。不管哪种情况， 重复的行都被删除，除非声明了 `ALL` 。噪声字 `DISTINCT` 可以用来明确的声明消除重复的行。请注意， `DISTINCT` 在这里是缺省的行为， 即使 `ALL` 是 `SELECT` 本身的缺省。 （参阅下面的*UNION 子句*、*INTERSECT 子句*、 *EXCEPT 子句*。）

8. 如果声明了 `ORDER BY` 子句，那么返回的行将按照指定的顺序排序。 如果没有给出 `ORDER BY` ，那么数据行是按照系统认为可以最快生成的顺序给出的。 （参阅下面的*ORDER BY 子句*。）

9. 如果给出了 `LIMIT` (或 `FETCH FIRST`)或 `OFFSET` 子句，那么 `SELECT` 语句只返回结果行的一个子集。 （参阅下面的*LIMIT 子句*。）

10. 如果声明了 `FOR UPDATE` ， `FOR NO KEY UPDATE` ， `FOR SHARE` 或 `FOR KEY SHARE` 子句，那么 `SELECT` 语句对并发的更新锁住选定的行。 （参阅下面的锁定子句子句。）

你必须对每个在 `SELECT` 命令中使用的字段有 `SELECT` 权限。 使用 `FOR NO KEY UPDATE` ， `FOR UPDATE` ， `FOR SHARE` 或 `FOR KEY SHARE` 还要求 `UPDATE` 权限 （至少选择每个表的一列）。

参数

`WITH` 子句

`WITH` 子句允许声明一个或多个可以在主查询中通过名字引用的子查询。 子查询在主查询期间有效的充当临时表或视图。每个子查询可以是 `SELECT` ， `VALUES` ， `INSERT` ， `UPDATE` 或 `DELETE` 语句。当在 `WITH` 中写一个数据修改语句时 （ `INSERT` ， `UPDATE` 或 `DELETE` ）， 通常包含一个 `RETURNING` 子句。 `RETURNING` 的输出而不是 语句修改的底层表的输出形成被主查询读取的临时表。如果省略了 `RETURNING` ， 该语句仍然执行，但是不会产生输出，所以不能作为一个表被主查询引用。

必须为每个 `WITH` 查询声明一个名字（没有模式修饰）。 可选的，可以指定字段名的列表；如果省略了，那么字段名从子查询中推断出。

如果声明了 `RECURSIVE` ，那么允许 `SELECT` 子查询通过名字引用它自己。 比如一个有下面形式的子查询

```
_non_recursive_term_ UNION [ ALL | DISTINCT ] _recursive_term_
```

递归的自我引用必须在 `UNION` 的右边出现。每个查询只允许一个递归的自我引用。 不支持递归的数据修改语句，但是可以在数据修改语句中使用递归的 `SELECT` 查询的结果。参阅Section 7.8获取一个例子。

`RECURSIVE` 的另外一个作用是 `WITH` 查询不需要排序： 一个查询可以引用另外一个稍后出现在列表中的查询。 （但是，循环引用或相互递归没有实现。） 没有 `RECURSIVE` ， `WITH` 查询只能引用 更早出现在 `WITH` 列表中的同层级的 `WITH` 查询。

`WITH` 查询的主要特性是他们只在主查询的每次执行中评估一次， 即使主查询引用了他们多次也是如此。特别的，保证数据修改语句只被执行一次， 不管主查询读取他们的所有或任意的输出。

主查询和 `WITH` 查询（理论上）同时执行。这意味着 `WITH` 中的数据修改语句的影响不能从查询的其他部分看到，除非读取它的 `RETURNING` 输出。如果两个这样的数据修改语句尝试修改相同的行，那么结果是未知的。

参阅Section 7.8获取额外的信息。

`FROM` 子句

`FROM` 子句为 `SELECT` 声明一个或者多个源表。如果声明了多个源表， 那么结果就是所有源表的笛卡儿积(交叉连接)。但是通常会添加一些条件（通过 `WHERE` ）， 把返回行限制成笛卡儿积的一个小的子集。

`FROM` 子句可以包括下列元素：

`_table_name_`

一个现存的表或视图的名字(可以有模式修饰)。如果声明了 `ONLY` ， 则只扫描该表； 否则，该表和所有其派生表(如果有的话)都被扫描。可以在表名后面跟一个 `*` 表示扫描所有其后代表。

`_alias_`

为那些包含别名的 `FROM` 项目取的别名。 别名用于缩写或者在自连接中消除歧义 (自连接中同一个表将扫描多次)。如果提供了别名，那么它就会完全隐藏表或者函数的实际名字； 比如，如果给出 `FROM foo AS f` ， 那么 `SELECT` 剩下的东西必须把这个 `FROM` 项按照 `f` 而不是 `foo` 引

用。 如果写了别名，也可以提供一个字段别名列表，这样可以替换表中一个或者多个字段的名字。

`_select_`

可以在 `FROM` 子句里出现一个子 `SELECT` 。 它的输出作用好像是为这条 `SELECT` 命令在其生存期里创建一个临时表。 请注意这个子 `SELECT` 必须用园括弧包围。并且必须给它一个别名。 当然，VALUES同样也可以在这里使用。

`_with_query_name_`

`WITH` 查询通过写它自己的名字来引用，就像查询的名字就是一个表的名字。 （实际上， `WITH` 查询为主查询隐藏了相同名字的任意实际表。 如果需要，可以通过模式限定表的名字来引用相同名字的实际表。） 别名可以用相同的方式提供给表。

`_function_name_`

函数(特别是那些返回结果集的函数)调用可以出现在 `FROM` 子句里。 这么做就好像在这个 `SELECT` 命令的生命期中，把函数的输出创建为一个临时表一样。 当然也可以使用别名。如果写了别名，还可以写一个字段别名列表， 为函数返回的复合类型的一个或多个属性提供名字替换。如果函数定义为返回 `record` 类型， 那么必须出现一个 `AS` 关键字或者别名，后面跟着一个形如 `(` `_column_name_` `_data_type_` [, ...])的字段定义列表。这个字段定义列表必须匹配函数返回的字段的实际数目和类型。

`_join_type_`

下列之一：

- `[ INNER ] JOIN`
- `LEFT [ OUTER ] JOIN`
- `RIGHT [ OUTER ] JOIN`
- `FULL [ OUTER ] JOIN`
- `CROSS JOIN`

必须为 `INNER` 和 `OUTER` 连接类型声明一个连接条件， 也就是 `NATURAL` , `ON` `_join_condition_` , or `USING (``_join_column_` [, ...]) 之一。 它们的含义见下文， 对于 `CROSS JOIN` 而言，这些子句都不能出现。

一个 `JOIN` 子句组合两个 `FROM` 项，为了方便我们将其作为"tables" 引用，尽管实际上他们可以是任意 `FROM` 条目的类型。必要时使用圆括弧以决定嵌套的顺序。 如果没有圆括弧， `JOIN` 从左向右嵌套。在任何情况下， `JOIN` 都比逗号分隔的 `FROM` 列表绑定得更紧。

`CROSS JOIN` 和 `INNER JOIN` 生成一个简单的笛卡儿积， 和你在 `FROM` 的顶层列出两个表的结果相同，但是受到连接条件（如果有）的限制。 `CROSS JOIN` 等效于 `INNER JOIN ON (TRUE)` ，也就是说， 没有被条件删除的行。这种连接类型只是符号上的方便，因为它们和你用简单的

`FROM` 和 `WHERE` 的效果一样。

`LEFT OUTER JOIN` 返回笛卡儿积中所有符合连接条件的行， 再加上左表中通过连接条件没有匹配右表行的那些行。这样，左边的行将扩展成生成表的全长， 方法是在那些右表对应的字段位置填上 NULL 。请注意，只在计算匹配的时候， 才使用 `JOIN` 子句的条件，外层的条件是在计算完毕之后施加的。

相应的， `RIGHT OUTER JOIN` 返回所有内连接的结果行， 加上每个不匹配的右边行(左边用 NULL 扩展)。这只是一个符号上的便利， 因为总是可以把它转换成一个 `LEFT OUTER JOIN` ，只要把左边和右边的输入对掉一下即可。

`FULL OUTER JOIN` 返回所有内连接的结果行，加上每个不匹配的左边行(右边用 NULL 扩展)，再加上每个不匹配的右边行(左边用 NULL 扩展)。

`ON` `_join_condition_`

一个生成 `boolean` 类型结果的表达式(类似 `WHERE` 子句)， 限定连接中那些行是匹配的。

`USING (` `_join_column_` [, ...])

一个形如 `USING ( a, b, ... )` 的子句，是 `ON left_table.a = right_table.a AND left_table.b = right_table.b ...` 的缩写。 同样， `USING` 蕴涵着每对等效字段中只有一个包含在连接输出中，而不是两个都输出的意思。

`NATURAL`

`NATURAL` 是一个 `USING` 列表的缩写，这个列表说的是两个表中同名的字段。

`LATERAL`

`LATERAL` 关键字可以放在一个子 `SELECT` `FROM` 项目的前面。 这允许子 `SELECT` 引用出现在 `FROM` 列表之前的 `FROM` 条目中的字段。 （没有 `LATERAL` ， 每个子 `SELECT` 独立评估并且以此不能交叉引用任何其他 `FROM` 条目。）

`LATERAL` 也可以放在一个函数调用 `FROM` 条目的前面， 但是这种情况下它是一个噪声字，因为函数表达式可以在任何情况下引用前面的 `FROM` 条目。

`LATERAL` 可以出现在 `FROM` 列表的顶层，或出现在 `JOIN` 树中。 在后面当前情况下，它也可以引用 `JOIN` 左侧和右侧的任意条目。

当 `FROM` 条目包含 `LATERAL` 交叉引用时，评估收益如下： 对于提供交叉引用字段的 `FROM` 条目的每一行，或提供字段的多个 `FROM` 条目的行集，使用该字段的行或行集值评估 `LATERAL` 条目。 结果行像往常一样和计算他们的行连接。这是从字段源表重复每行或行集。

字段源表必须 `INNER` 或 `LEFT` 连接 `LATERAL` 条目， 其他为 `LATERAL` 条目计算每个行集的行集将不会是明确定义的行集。 因此，尽管一个构造（如 `_X_` RIGHT JOIN LATERAL `_Y_` ） 在语法结构上合法，并不实际上允许 `_Y_` 引用 `_X_` 。

`WHERE` 子句

可选的 `WHERE` 条件有如下常见的形式：

```
WHERE _condition_
```

这里 `_condition_` 可以是任意生成类型为 `boolean` 的表达式。任何不满足这个条件的行都会从输出中删除。 如果一个行的数值代入到条件中计算出来的结果为真，那么该行就算满足条件。

`GROUP BY` 子句

可选的 `GROUP BY` 子句的一般形式

```
GROUP BY _expression_ [, ...]
```

它将把所有在组合表达式上拥有相同值的行压缩成一行。 `_expression_` 可以是一个输入字段名字， 或者是一个输出字段(`SELECT` 列表项)的名字或序号， 或者也可以是任意输入字段组成的表达式。在有歧义的情况下， 一个 `GROUP BY` 的名字将被解释成输入字段的名字，而不是输出字段的名字。

如果使用了聚集函数，那么就会对每组中的所有行进行计算并生成一个单独的值 (而如果没有 `GROUP BY` ，那么聚集将对选出来的所有行计算出一个单独的值)。 如果出现了 `GROUP BY` ，那么 `SELECT` 列表表达式中再引用那些没有分组的字段就是非法的，除非放在聚集函数里，或未分组的字段函数上依赖于分组的字段，因为对于未分组的字段，可能会返回多个数值。如果分组的字段（或它们的一个子集）是包含未分组字段的表的主键，那么存在一个函数依赖。

`HAVING` 子句

可选的 `HAVING` 子句有如下形式：

```
HAVING _condition_
```

这里 `_condition_` 与为 `WHERE` 子句里声明的相同。

`HAVING` 去除了一些不满足条件的组行。它与 `WHERE` 不同： `WHERE` 在使用 `GROUP BY` 之前过滤出单独的行， 而 `HAVING` 过滤由 `GROUP BY` 创建的行。 在 `_condition_` 里引用的每个字段都必须无歧义地引用一个分组的行，除非引用出现在一个聚集函数里。

`HAVING` 的出现把查询变成一个分组的查询，即使没有 `GROUP BY` 子句也这样。这一点和那些包含聚集函数但没有 `GROUP BY` 子句的查询里发生的事情是一样的。 所有选取的行都被认为会形成一个单一的组，而 `SELECT` 列表和 `HAVING` 子句只能从聚集函数里面引用表的字段。 这样的查询在 `HAVING` 条件为真的时候将发出一个行，如果为非真，则返回零行。

`WINDOW` 子句

`WINDOW` 子句的一般形式是

```
WINDOW _window_name_ AS ( _window_definition_ ) [, ...]
```

这里的 `_window_name_` 是可以在 `OVER` 子句中引用的名字或随后的窗口定义，这里的 `_window_definition_` 是

```
[ _existing_window_name_ ]
[ PARTITION BY _expression_ [, ...] ]
[ ORDER BY _expression_ [ ASC | DESC | USING _operator_ ] [ NULLS { FIRST | LAST } ] [, .
[ _frame_clause_ ]
```

如果声明了 `_existing_window_name_` ， 那么必须引用一个在 `WINDOW` 列表中出现的更早的项；新的窗口从这个项中拷贝分区子句， 和排序子句（如果有）。在这种情况下，这个新的窗口不能声明他自己的 `PARTITION BY` 子句， 但是如果拷贝的窗口没有排序子句的话他可以声明 `ORDER BY` 子句。 新窗口总是使用他自己的框架子句；拷贝的窗口必须不能声明框架子句。

`PARTITION BY` 列表的元素的以与*GROUP BY 子句* 的元素相同的方式来解释，除了他们总是简单的表达式并且从不是一个输出列的名称或者编号。 另外一个差异是这些表达式可以包含聚集函数调用，而在常规的 `GROUP BY` 子句中这是不允许的。 在这里允许是因为开窗在分组和聚集之后发生。

同样的， `ORDER BY` 列表的元素以与*ORDER BY 子句* 的元素相同的方式来解释，除了这个表达式总是作为简单的表达式并且从不是一个输出列的名称或者编号。

可选的 `_frame_clause_` 为依赖于框架的窗口函数（不是所有）定义 窗口框架。窗口框架是查询中的每行（称为当前行）的一组相关行。 `_frame_clause_` 可以是下列之一

```
[ RANGE | ROWS ] _frame_start_
[ RANGE | ROWS ] BETWEEN _frame_start_ AND _frame_end_
```

这里的 `_frame_start_` 和 `_frame_end_` 可以是下列之一

```
UNBOUNDED PRECEDING
_value_ PRECEDING
CURRENT ROW
_value_ FOLLOWING
UNBOUNDED FOLLOWING
```

如果省略了 `_frame_end_` ，那么它的缺省是 `CURRENT ROW` 。 限制是 `_frame_start_` 不能是 `UNBOUNDED FOLLOWING` ， `_frame_end_` 不能是 `UNBOUNDED PRECEDING` ， `_frame_end_` 选项在上面的列表中不能比 `_frame_start_` 选项出现的早，例如 `RANGE BETWEEN CURRENT ROW AND _value_` PRECEDING是不允许的。

缺省框架选项是 `RANGE UNBOUNDED PRECEDING` ，这与 `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 相同；它将框架设置为分区中的所有行，在 `ORDER BY` 序列中是从当前行的最后一个元素开始（这意味着若无 `ORDER BY` 则是所有行）。 通常， `UNBOUNDED PRECEDING` 表示框架从分区的第一行开始，类似的 `UNBOUNDED FOLLOWING` 表示框架以分区的最后一行结束（不管是 `RANGE` 还是 `ROWS` 模式）。 在 `ROWS` 模式， `CURRENT ROW` 意味着框架以当前行开始或结束；但是在 `RANGE` 模式，意味着框架以当前行在 `ORDER BY` 序列中的第一个或最后一个元素开始或结束。 `_value_ PRECEDING` 和 `_value_ FOLLOWING` 子句目前只允许在 `ROWS` 模式。 他们表明框架以当前行之前或者之后许多行开始或者结束。 `_value_` 必须是一个不包含任何变量、聚集函数或者窗口函数的整型表达式。该值不能为空或者负值； 但可以为0，并且这时选择当前行本身。

注意如果 `ORDER BY` 排序不能唯一地排列行，那么 `ROWS` 选项可能产生不可预测的结果。 `RANGE` 选项是为了确保 `ORDER BY` 序列中的对等的行能得到同等对待；任何两个对等行将会都在或者都不在框架中。

一个 `WINDOW` 语句的目的是指定出现在查询的*SELECT 列表* 或者*ORDER BY 子句*中的*window*函数的行为。 这些函数可以在其 `OVER` 子句中通过名称引用 `WINDOW` 子句条目。 一个 `WINDOW` 子句条目不需要在任何地方都引用；若它不在查询中使用，它将被忽略。 可以使用窗口程序而根本不需要任何 `WINDOW` 子句， 因为一个窗口函数调用可以直接在其 `OVER` 子句中指定其窗口定义。然而， `WINDOW` 子句会在多个窗口函数需要相同窗口定义时保存输入。

窗口函数在Section 3.5, Section 4.2.8, 和 Section 7.2.4中有详细描述。

`SELECT` 列表

`SELECT` 列表(在 `SELECT` 和 `FROM` 关键字之间的部分) 声明组成 `SELECT` 语句的输出行的表达式。这些表达式可以 (并且通常也会)引用在 `FROM` 子句里面计算出来的字段。

就像在一个表中，一个 `SELECT` 的每个输出列都有一个名称。 在一个简单的 `SELECT` 中，该名称仅用于标记显示的列， 但当 `SELECT` 是一个较大查询的子查询时，名称被较大查询视为子查询产生的虚表的字段名。 为了指定用于输出列的名称，要在列表达式后写 `AS _output_name_` 。 （您可以省略 `AS` ， 但只有当所需的输出名称不匹配任何PostgreSQL关键字时 (请参阅Appendix C)。为了防止将来可能的关键字添加， 建议您要么写 `AS` 要么用双引号引起输出名称。）如果你不指定一个字段名， PostgreSQL会自动选择一个名称。如果字段的表达式是一个简单的列引用， 那么选择的名称与字段名相同；在更复杂的情况下，可能会使用一个函数或类型名， 或系统会依赖于一个类似 `?column?` 的生成名。

一个输出列的名称可以用来参考 `ORDER BY` 和 `GROUP BY` 子句中的字段的值， 而不是在 `WHERE` 或者 `HAVING` 子句中的；反而您必须在那里写出表达式。

除了表达式，也可以在输出列表中使用 `*` 表示所有字段。 还可以用 `_table_name_` .* 作为来自该表的所有字段的缩写。这些情况下用 `AS` 指定新名称是不可能的； 输出列的名称将会与表列的名称相同。

`DISTINCT` 子句

如果声明了 `SELECT DISTINCT` ，那么就从结果集中删除所有重复的行 (每个有重复的组都保留一行)。 `SELECT ALL` 声明相反的作用：所有行都被保留(这是缺省)。

`SELECT DISTINCT ON (` `_expression_` [, ...]) 只保留那些在给出的表达式上运算出相同结果的行集合中的第一行。 `DISTINCT ON` 表达式是使用与 `ORDER BY` 相同的规则进行解释的（见上文）。请注意， 除非使用了 `ORDER BY` 来保证需要的行首先出现，否则，"第一行" 是不可预测的。比如：

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果没有使用 `ORDER BY` 来强制对每个地点的时间值进行降序排序，那么就会得到每个地点的不知道什么时候的报告。

`DISTINCT ON` 表达式必须匹配最左边的 `ORDER BY` 表达式。 `ORDER BY` 子句将通常包含额外的表达式来判断每个 `DISTINCT ON` 组里面需要的行的优先级。

`UNION` 子句

`UNION` 子句的一般形式是：

```
_select_statement_ UNION [ ALL | DISTINCT ] _select_statement_
```

这里的 `_select_statement_` 是任意没有 `ORDER BY` , `LIMIT` , `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` , 或 `FOR KEY SHARE` 子句的 `SELECT` 语句。 如果用圆括弧包围， `ORDER BY` 和 `LIMIT` 可以附着在子表达式里。 如果没有圆括弧，这些子句将交给 `UNION` 的结果使用，而不是给它们右边的输入表达式。

`UNION` 操作符计算那些涉及到的所有 `SELECT` 语句返回的行的结果联合。一个行如果至少在两个结果集中的一个里面出现， 那么它就会在这两个结果集的集合联合中。两个作为 `UNION` 直接操作数的 `SELECT` 必须生成相同数目的字段， 并且对应的字段必须有兼容的数据类型。

缺省的 `UNION` 结果不包含任何重复的行，除非声明了 `ALL` 选项。 `ALL` 制止了消除重复的动作。因此， `UNION ALL` 通常比 `UNION` 明显要快，可能的情况下尽量使用 `ALL` 。 `DISTINCT` 可以明确的指定消除重复行的缺省行为。

同一个 `SELECT` 语句中的多个 `UNION` 操作符是从左向右计算的， 除非用圆括弧进行了标识。

目前， `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` 和 `FOR KEY SHARE` 不能在 `UNION` 的结果或输入中声明。

`INTERSECT` 子句

`INTERSECT` 子句的一般形式是：

```
_select_statement_ INTERSECT [ ALL | DISTINCT ] _select_statement_
```

`_select_statement_` 是任何不带 `ORDER BY` , `LIMIT` , `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` , 或 `FOR KEY SHARE` 子句的 `SELECT` 语句。

`INTERSECT` 计算涉及的 `SELECT` 语句返回的行集合的交集。 如果一个行在两个结果集中都出现，那么它就在两个结果集的交集中。

`INTERSECT` 的结果不包含任何重复行，除非你声明了 `ALL` 选项。 用了 `ALL` 以后，一个在左边的表里有 `_m_` 个重复而在右边表里有 `_n_` 个重复的行将在结果集中出现min(`_m_` , `_n_`)次。 `DISTINCT` 可以明确的指定消除重复行的缺省行为。

除非用圆括号指明顺序，同一个 `SELECT` 语句中的多个 `INTERSECT` 操作符是从左向右计算的。 `INTERSECT` 比 `UNION` 绑定得更紧，也就是说 `A UNION B INTERSECT C` 将理解成 `A UNION (B INTERSECT C)` 。

目前，不能给 `INTERSECT` 的结果或者任何 `INTERSECT` 的输入声明 `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` 和 `FOR KEY SHARE` 。

`EXCEPT` 子句

`EXCEPT` 子句有如下的通用形式：

```
_select_statement_ EXCEPT [ ALL | DISTINCT ] _select_statement_
```

`_select_statement_` 是任何没有 `ORDER BY` , `LIMIT` , `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` , 或 `FOR KEY SHARE` 子句的 `SELECT` 表达式。

`EXCEPT` 操作符计算存在于左边 `SELECT` 语句的输出而不存在于右边 `SELECT` 语句输出的行。

`EXCEPT` 的结果不包含任何重复的行，除非声明了 `ALL` 选项。 使用 `ALL` 时，一个在左边表中有 `_m_` 个重复而在右边表中有 `_n_` 个重复的行将在结果中出现max(`_m_` - `_n_` ,0)次。 `DISTINCT` 可以明确的指定消除重复行的缺省行为。

除非用圆括弧指明顺序，否则同一个 `SELECT` 语句中的多个 `EXCEPT` 操作符是从左向右计算的。 `EXCEPT` 和 `UNION` 的绑定级别相同。

目前，不能给 `EXCEPT` 的结果或者任何 `EXCEPT` 的输入声明 `FOR NO KEY UPDATE` , `FOR UPDATE` , `FOR SHARE` 和 `FOR KEY SHARE` 子句。

`ORDER BY` 子句

可选的 `ORDER BY` 子句有下面的一般形式：

```
ORDER BY _expression_ [ ASC | DESC | USING _operator_ ] [ NULLS { FIRST | LAST } ] [, ...
```

`ORDER BY` 子句导致结果行根据指定的表达式进行排序。 如果根据最左边的表达式，两行的结果相同，那么就根据下一个表达式进行比较， 依此类推。如果对于所有声明的表达式他们都相同，那么按随机顺序返回。

每个 `_expression_` 可以是一个输出字段 (`SELECT` 列表项)的名字或者序号，或者也可以是用输入字段的数值组成的任意表达式。

序数指的是输出字段按顺序(从左到右)的位置。这个特性可以对没有唯一名称的字段进行排序。 这不是必须的，因为总是可以通过 `AS` 子句给一个要输出的字段赋予一个名称。

在 `ORDER BY` 里还可以使用任意表达式，包括那些没有出现在 `SELECT` 输出列表里面的字段。因此下面的语句现在是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这个特性的一个局限就是应用于 `UNION` , `INTERSECT` , `EXCEPT` 子句的结果的 `ORDER BY` 子句只能在一个输出字段名或者数字上声明， 而不能在一个表达式上声明。

如果一个 `ORDER BY` 表达式是一个简单名称，同时匹配结果字段和输入字段， `ORDER BY` 将把它解释成结果字段名称。这和 `GROUP BY` 在同样情况下做的选择正相反。这样的不一致是用来和 SQL 标准兼容的。

可以给 `ORDER BY` 子句里每个字段加一个可选的 `ASC` (升序，缺省) 或 `DESC` (降序)关键字。还可以在 `USING` 子句里声明一个排序操作符来实现排序。排序操作符必须小于或大于某些B-tree操作符族的成员。 `ASC` 等效于使用 `USING <` 而 `DESC` 等效于使用 `USING >` 。 但是一个用户定义类型的创建者可以明确定义缺省的排序顺序，并且可以使用其他名称的操作符。

如果指定 `NULLS LAST` ，空值会在所有非空值之后排序；如果指定 `NULLS FIRST` ，空值会在所有非空值之前排序。如果两者均未指定， 当指定 `ASC` 或缺省时，默认反应时是 `NULLS LAST` ，并且当指定 `DESC` 时，默认反应时是 `NULLS FIRST` （因此， 默认地认为空是大于非空的）。当指定 `USING` 时， 默认空排序依赖于操作符是小于还是大于操作符。

请注意排序选项仅适用于他们遵循的表达式；例如 `ORDER BY x, y DESC` 不意味着与 `ORDER BY x DESC, y DESC` 相同。

字符类型的数据是按照应用于被排序的字段的排序规则排序的。可以在需要时通过在 `_expression_` 中包含 `COLLATE` 子句覆盖，例如 `ORDER BY mycolumn COLLATE "en_US"` 。更多信息请参阅 Section 4.2.10和Section 22.2。

`LIMIT` 子句

`LIMIT` 子句由两个独立的子句组成：

```
LIMIT { _count_ | ALL }
OFFSET _start_
```

`_count_` 声明返回的最大行数， 而 `_start_` 声明开始返回行之前忽略的行数。 如果两个都指定了，那么在开始计算 `_count_` 个返回行之前将先跳过 `_start_` 行。

如果 `_count_` 表达式评估为NULL， 它被当做 `LIMIT ALL` ， 也就是，没有限制。如果 `_start_` 评估为NULL，他与 `OFFSET 0` 相同对待。

SQL:2008引入了一个不同的语法来达到相同的效果，这也是PostgreSQL支持的。这是：

```
OFFSET _start_ { ROW | ROWS }
FETCH { FIRST | NEXT } [ _count_ ] { ROW | ROWS } ONLY
```

在该语法中，为 `_start_` 或 `_count_` 提供除简单整型常量之外的东西时，你必须写圆括号。如果 `_count_` 在 `FETCH` 子句中省略了，它默认为1。 `ROW` 和 `ROWS` 以及 `FIRST` 和 `NEXT` 是不影响这些子句的效果的干扰词。 根据该标准，若两个都存在则 `OFFSET` 子句必须在 `FETCH` 子句之前出现； 但是PostgreSQL的要求更为宽松并且允许任意一种顺序。

使用 `LIMIT` 的一个好习惯是使用一个 `ORDER BY` 子句把结果行限制成一个唯一的顺序。 否则你会得到无法预料的结果子集，你可能想要第十行到第二十行，但是是以什么顺序的第十行到第二十行？ 除非你声明 `ORDER BY` ，否则你不知道什么顺序。

查询优化器在生成查询规划时会把 `LIMIT` 考虑进去，所以你很有可能因给出的 `LIMIT` 和 `OFFSET` 值不同而得到不同的规划(生成不同的行序)。 因此用不同的 `LIMIT` / `OFFSET` 值选择不同的查询结果的子集 将不会产生一致的结果，除非你用 `ORDER BY` 强制生成一个可预计的结果顺序。这可不是 bug ；这是 SQL 生来的特点， 因为除非用了 `ORDER BY` 约束顺序，SQL 不保证查询生成的结果有任何特定的顺序。

如果没有一个 `ORDER BY` 来强制选择一个确定性子集，那么重复执行相同的 `LIMIT` 查询返回不同的表行的子集甚至都是可能的。同样， 这不是一个漏洞；结果的确定在这种情况下没法保证。

锁定子句

`FOR UPDATE` , `FOR NO KEY UPDATE` , `FOR SHARE` 和 `FOR KEY SHARE` 是 锁定子句;他们影响 `SELECT` 如何从表中锁定行作为获得的行。

锁定子句的一般形式：

```
FOR _lock_strength_ [ OF _table_name_ [, ...] ] [ NOWAIT ]
```

这里的 `_lock_strength_` 可以是下列之一：

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

`FOR UPDATE` 令那些被 `SELECT` 检索出来的行被锁住， 就像要更新一样。这样就避免它们在当前事务结束前被其它事务修改或者删除；也就是说， 其它企图 `UPDATE` ， `DELETE` , `SELECT FOR UPDATE` , `SELECT FOR NO KEY UPDATE` , `SELECT FOR SHARE` 或 `SELECT FOR KEY SHARE` 这些行的事务将被阻塞，直到当前事务结束。 `FOR UPDATE` 锁模式也可以通过在一个行上 `DELETE` 或在特定的字段上修改值的 `UPDATE` 获得。目前，为 `UPDATE` 情况考虑的字段设置是那些有唯一索引并且可以用于外键的 （所以不考虑局部索引和表达式索引）， 但是这个可能会在将来改变。同样， 如果一个来自其它事务的 `UPDATE` ， `DELETE` , `SELECT FOR UPDATE` 已经锁住了某个或某些选定的行， `SELECT FOR UPDATE` 将等到那些事务结束，并且将随后锁住并返回更新的行(或者不返回行，如果行已经被删除)。 但是，在 `REPEATABLE READ` 或 `SERIALIZABLE` 事务内部， 如果在事务开始时要被锁定的行已经改变了，那么将抛出一个错误。更多的讨论参阅Chapter 13。

`FOR NO KEY UPDATE` 的行为类似，只是获得的锁比较弱：这个锁将不锁定 尝试在相同的行上获得锁的 `SELECT FOR KEY SHARE` 命令。 这个锁模式也可以通过任何不争取 `FOR UPDATE` 锁的 `UPDATE` 获得。

`FOR SHARE` 的行为类似，只是它在每个检索出来的行上要求一个共享锁， 而不是一个排它锁。一个共享锁阻塞其它事务在这些行上执行 `UPDATE` ， `DELETE` ， `SELECT FOR UPDATE` 或 `SELECT FOR NO KEY UPDATE` ， 却不阻止他们执行 `SELECT FOR SHARE` 或 `SELECT FOR KEY SHARE` 。

`FOR KEY SHARE` 的行为类似于 `FOR SHARE` ， 只是锁比较弱： 阻塞 `SELECT FOR UPDATE` 但不阻塞 `SELECT FOR NO KEY UPDATE` 。 一个共享键块阻塞其他事务执行 `DELETE` 或任意改变键值的 `UPDATE` ， 但是不阻塞其他 `UPDATE` ， 也不阻止 `SELECT FOR NO KEY UPDATE` ,

`SELECT FOR SHARE`，或 `SELECT FOR KEY SHARE` 。

为了避免操作等待其它事务提交，使用 `NOWAIT` 选项。如果被选择的行不能立即被锁住， 那么语句将会立即汇报一个错误，而不是等待。请注意， `NOWAIT` 只适用于行级别的锁， 要求的表级锁 `ROW SHARE` 仍然以通常的方法进行(参阅Chapter 13)。 如果需要申请表级别的锁同时又不等待，那么你可以使用LOCK的 `NOWAIT` 选项。

如果在锁定子句中明确指定了表名字，那么将只有这些指定的表被锁定，其他在 `SELECT` 中使用的表将不会被锁定。 一个其后不带表列表的锁定子句将锁定该声明中所有使用的表。 如果锁定子句应用于一个视图或者子查询，它同样将锁定所有该视图或子查询中使用到的表。 但是这些子句不适用于被主查询引用的 `WITH` 查询。 如果你想要行锁定发生在 `WITH` 查询中，那么在 `WITH` 查询中指定锁定子句。

多个锁定子句可以用于为不同的表指定不同的锁定模式。如果一个表出同时出现 (或隐含同时出现)在多个锁定子句中，那么将看做只被最强的那个声明了处理。 类似的，如果影响一个表的任意子句中出现了 `NOWAIT` ， 那么该表将按照 `NOWAIT` 处理。

锁定子句不能在那些无法使用独立的表行清晰标识返回行的环境里使用；比如， 它不能和聚集一起使用。

当锁定子句出现在 `SELECT` 查询的顶层时，锁定的行恰好是被查询返回的行； 在连接查询的情况下，锁定的行是那些有助于返回连接的行。另外， 满足查询条件的行作为查询快照将被锁定，尽管如果在快照后他们被更新不再满足查询条件 就不再被返回。如果使用了 `LIMIT` ，那么一旦返回足够的行满足限制锁定将停止 （但是请注意，通过 `OFFSET` 跳过的行将被锁定）。 相似的，如果锁定子句用于游标查询， 只有实际抓取到的行或通过游标跳过的行将被锁定。

当锁定子句出现在子 `SELECT` 中时，锁定的行是那些通过子查询返回到外查询的行。 这些包含的行可能比单独检查子查询时给出的行更少，因为外查询的条件可能会用来优化子查询的执行。 例如：

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

将只锁定 `col1 = 5` 的行，即使那个条件不是子查询中的原文。

以前的版本未能保持锁，通过一个稍后的保存点来改善。例如，这段代码：

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

`FOR UPDATE` 锁将会在 `ROLLBACK TO` 之后无法保持。这在版本9.3中已经修复了。

Caution

一个 `SELECT` 命令运行在 `READ COMMITTED` 事务隔离级别和使用 `ORDER BY` 和一个锁定子句返回顺序混乱的行是可能的。这是因为 `ORDER BY` 先生效。命令排序结果，但是可能会在其中一行或多行上获取锁的时候被阻塞。一旦 `SELECT` 的阻塞被解除后，某些顺序字段值可能被修改，导致这些行混乱了（尽管他们还是原先字段值的顺序）。 可以在需要的时候通过在子查询中放置 `FOR UPDATE/SHARE` 子句来避开，例如

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

请注意这将导致锁定 `mytable` 的所有行，而顶层的 `FOR UPDATE` 将会实际上仅锁住返回行。这可能会产生一个显著的性能差异，尤其是如果 `ORDER BY` 与 `LIMIT` 或者其他限制结合。仅当顺序列的并发更新是预期的并且需要一个严格的排序结果时， 该技术才是建议使用的。

在 `REPEATABLE READ` 或 `SERIALIZABLE` 事务隔离级别， 这可能会导致一个序列化失败（ `SQLSTATE` 为 `'40001'` ）， 所以在这些隔离级别下接收次序混乱的行是不可能的。 |

`TABLE` 命令

命令：

```
TABLE _name_
```

完全等价于：

```
SELECT * FROM _name_
```

它可以用作复杂查询中的一部分的一个顶级的命令或者一个节省空间的语法变体。

例子

将表 `films` 和表 `distributors` 连接在一起：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

统计用 `kind` 分组的每组电影的长度 `len` 总和：

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

统计用 `kind` 分组的每组电影的长度 `len` 总和不足五小时的组：

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';

   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

下面两个例子是根据第二列(`name`)的内容对单独的结果排序的相同的方法：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

下面这个例子演示如何获得表 `distributors` 和 `actors` 的连接， 只将每个表中以字母 W 开头的取出来。因为只取了不重复的行，所以关键字 `ALL` 被省略了：

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...

SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

这个例子显示了如何在 `FROM` 子句中使用函数，包括带有和不带字段定义列表的。

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

这个例子显示了如何使用一个简单的 `WITH` 子句：

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

请注意 `WITH` 查询只评估一次，所以我们获得两组相同的三个随机值。

这个例子使用 `WITH RECURSIVE` 从一个只显示直接属下的表中找到所有职员Mary的属下（直接或间接），和他们的间接级别，

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

请注意递归查询的典型形式：一个初始条件，紧接着是 `UNION`， 然后是查询的递归部分。确定查询的递归部分最终将不会返回元组，否则查询将无限循环下去。 （请参阅Section 7.8获取更多示例。）

这个例子使用 `LATERAL` 为 `manufacturers` 表的每行应用一个设置返回函数 `get_product_names()` ：

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

当前没有任何产品的制造商将不会出现在结果中，因为这是一个内连接。 如果我们希望在结果中包括这种制造商的名字，我们可以这样做：

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

兼容性

`SELECT` 语句和 SQL 标准兼容。但是还有一些扩展和一些缺少的特性。

省略 `FROM` 子句

PostgreSQL允许在一个查询里省略 `FROM` 子句。 它的最直接用途就是计算简单的常量表达式的结果：

```
SELECT 2+2;

 ?column?
----------
        4
```

其它有些SQL数据库不能这么做，除非引入一个单行的伪表做为 `SELECT` 的数据源。

请注意，如果没有声明 `FROM` 子句，那么查询不能引用任何数据库表。 比如，下面的查询是非法的：

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL 8.1 之前的版本支持这种形式的查询， 为查询里引用的每个表都增加一个隐含的条目到 `FROM` 子句中。 现在这个不再是允许的了。

省略 `AS` 关键字

在SQL标准中，每当新列名称是一个有效的列名时（也就是，与任意保留关键字都不同），可选的关键字 `AS` 可以在输出列名之前省略。PostgreSQL 限制略多一些：不管是保留还是不保留，如果新列名匹配任何关键字， `AS` 是必要的。 建议的做法是使用 `AS` 或者双括号括起输出列名称，以阻止与将来补充的关键字有任何可能的冲突。

在 `FROM` 项中，标准和PostgreSQL都允许 `AS` 在一个无限制关键字别名之前省略。但是这对输出列名是不切实际的，因为语法的歧义。

`ONLY` 和继承

SQL标准需要在写 `ONLY` 时括号括起表名，例如 `SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...`。 PostgreSQL认为括号是可选的。

PostgreSQL允许写一个尾随的 `*` 以明确指定包括子表的非 `ONLY` 行为。标准不允许这样。

（该点同样适用于所有支持 `ONLY` 选项的SQL命令。）

`FROM` 中的函数调用

PostgreSQL允许把函数调用直接写作 `FROM` 列表的一个成员。 在SQL标准中应该有必要在子 `SELECT` 中包含这样一个函数；也就是， 该语法 `FROM` `_func_` (...) `_alias_` 等价于 `FROM LATERAL (SELECT` `_func_` (...)) `_alias_` 。 请注意 `LATERAL` 被认为是隐含的；这是因为标准要求 `LATERAL` 语义为 一个在 `FROM` 中的 `UNNEST()` 条目。PostgreSQL 对待 `UNNEST()` 和其他设置返回函数相同。

`GROUP BY` 和 `ORDER BY` 里可用的命名空间

在 SQL-92 标准里， `ORDER BY` 子句只能使用输出字段名或者编号， 而 `GROUP BY` 子句只能用基于输入字段名的表达式。PostgreSQL 对这两个子句都进行了扩展，允许另外一种选择(但是如果存在歧义，则使用标准的解释)。 PostgreSQL还允许两个子句声明任意的表达式。 请注意在表达式中出现的名字总是被当作输入字段名，而不是输出字段名。

SQL:1999 以及之后的一个略微不同的定义并不能和 SQL-92 完全向前兼容。不过， 在大多数情况下，PostgreSQL将把一个 `ORDER BY` 或 `GROUP BY` 表达式解析成为 SQL:1999 制定的那样。

函数依赖

PostgreSQL只在表的主键包含在 `GROUP BY` 列表中时识别函数依赖（允许在 `GROUP BY` 中省略字段）。 SQL标准指定应该被识别的附加条件。

`WINDOW` 子句限制

SQL标准为窗口 `_frame_clause_` 提供了附加选项。 PostgreSQL目前仅支持上面列出的选项。

`LIMIT` 和 `OFFSET`

子句 `LIMIT` 和 `OFFSET` 是PostgreSQL 特定的语法，也是MySQL使用的。SQL:2008标准引入了 `OFFSET ... FETCH {FIRST|NEXT} ...` 获取相同的功能， 如上面的*LIMIT 子句*子句所示。 该语法也被IBM DB2使用。（为Oracle 所写的应用程序通常使用一个涉及自动生成的 `rownum` 列的工作区， 来实现这些子句的效果，这在PostgreSQL中是不可用的。）

`FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, `FOR KEY SHARE`

尽管 `FOR UPDATE` 出现在了SQL标准中，但是标准只允许它作为 `DECLARE CURSOR` 的一个选项。 PostgreSQL允许它出现在任意 `SELECT` 查询和子 `SELECT` 查询中，但是这是一个扩展。 `FOR NO KEY UPDATE`, `FOR SHARE`, `FOR KEY SHARE` 变体和 `NOWAIT` 选项， 没有出现在标准中。

`WITH` 中的数据修改语句

PostgreSQL允许 `INSERT`, `UPDATE`, `DELETE` 用作 `WITH` 查询。这在SQL标准中是没有的。

非标准的子句

子句 `DISTINCT ON` 在SQL标准中未定义。

SELECT INTO

Name

SELECT INTO -- 从一条查询的结果中定义一个新表

Synopsis

```
[ WITH [ RECURSIVE ] _with_query_ [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( _expression_ [, ...] ) ] ]
    * | _expression_ [ [ AS ] _output_name_ ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] _new_table_
    [ FROM _from_item_ [, ...] ]
    [ WHERE _condition_ ]
    [ GROUP BY _expression_ [, ...] ]
    [ HAVING _condition_ [, ...] ]
    [ WINDOW _window_name_ AS ( _window_definition_ ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] _select_ ]
    [ ORDER BY _expression_ [ ASC | DESC | USING _operator_ ] [ NULLS { FIRST | LAST } ]
    [ LIMIT { _count_ | ALL } ]
    [ OFFSET _start_ [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ _count_ ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF _table_name_ [, ...] ] [ NOWAIT ] [...] ]
```

描述

`SELECT INTO` 从一个查询中创建一个新表,并且将查询到的数据插入到新表中。数据并不返回给客户端， 这一点和普通的 `SELECT` 不同。新表的字段具有和 `SELECT` 的输出字段相同的名字和数据类型。

参数

`TEMPORARY` 或 `TEMP`

如果声明了这个关键字，那么该表是作为一个临时表创建的。 请参考CREATE TABLE获取细节。

`UNLOGGED`

如果指定了，那么表作为一个非日志表创建。参阅CREATE TABLE获取详细信息。

`_new_table_`

要创建的表的名字(可以有模式修饰)。

所有其它参数都在SELECT中有详细描述。

注意

CREATE TABLE AS的作用和 `SELECT INTO` 类似。 建议使用 `CREATE TABLE AS` 语法。 实际上，它是不能在ECPG或PL/pgSQL中使用的， 因为它们对 `INTO` 子句的解释是不同的。而且， `CREATE TABLE AS` 提供了 `SELECT INTO` 所提供功能的超集。

在PostgreSQL 8.1以前， `SELECT INTO` 创建的表总是缺省包含 OID 。 到了PostgreSQL 8.1，这不再是缺省了，要想在新表中包含 OID ， 可以打开default_with_oids配置参数.命令 `CREATE TABLE AS` 可以使用 `WITH OIDS` 子句包含oid列。

例子

创建一个新表 `films_recent` ，它的值包含来自 `films` 的最近的条目：

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

兼容性

SQL 标准用 `SELECT INTO` 表示选取数值到一个宿主程序的标量变量中， 而不是创建一个新表。这种用法实际上就是在ECPG (Chapter 33) 和PL/pgSQL(Chapter 40) 里的用途。PostgreSQL用 `SELECT INTO` 创建表是历史原因。在新代码里最好使用 `CREATE TABLE AS` 实现这个目的。

又见

CREATE TABLE AS

SET

Name

SET -- 修改运行时参数

Synopsis

```
SET [ SESSION | LOCAL ] _configuration_parameter_ { TO | = } { _value_ | '_value_' | DEFA
SET [ SESSION | LOCAL ] TIME ZONE { _timezone_ | LOCAL | DEFAULT }
```

描述

`SET` 命令修改运行时配置参数。许多在Chapter 18 里面列出的运行时参数可以用 `SET` 在运行时设置。但是有些要求使用超级用户权限来修改， 而其它有些则在服务器或者会话开始之后不能修改。请注意 `SET` 只影响当前会话使用的数值。

如果 `SET` （或相等的 `SET SESSION` ）是在一个稍后退出的事务里发出的， 那么 `SET` 命令的效果将在事务回滚之后消失。一旦包围它的事务提交， 那么其效果将持续到会话结束，除非被另外一个 `SET` 覆盖。

不管是否提交， `SET LOCAL` 的效果只持续到当前事务结束。 一个特例是在一个事务里面的 `SET` 后面跟着一个 `SET LOCAL` ： 在事务结束之前只能看到 `SET LOCAL` 的数值，但是之后(如果事务提交)， 则是 `SET` 的值生效。

`SET` 或 `SET LOCAL` 的影响也可以通过回滚到一个命令之前的保存点取消。

如果 `SET LOCAL` 在一个对相同变量有 `SET` 选项的函数内使用 (请参阅CREATE FUNCTION)， `SET LOCAL` 命令的影响在函数退出时消失；也就是说，函数调用时起作用的值不论如何都会还原。 这允许 `SET LOCAL` 用来对函数内的一个参数进行动态和反复修改， 尽管仍然方便使用 `SET` 选项来保存和存储调用程序的值。然而， 规则的 `SET` 命令重写任何函数内的 `SET` 选项；其影响将会持续，除非回滚。

> **Note:** 在PostgreSQL 8.0到8.2版本中，一个 `SET LOCAL` 的影响将通过释放一个较早的保存点被取消，或者通过从一个PL/pgSQL 异常块成功退出。该选项已被更改，因为它被认为是非直观的。

参数

`SESSION`

声明这个命令只对当前会话起作用。如果 `SESSION` 或 `LOCAL` 都没出现， 那么这个是缺省。

`LOCAL`

声明该命令只在当前事务中有效。在 `COMMIT` 或者 `ROLLBACK` 之后， 会话级别的设置将再次生效。请注意如果在 `BEGIN` 块之外运行， 那么 `SET LOCAL` 将表现出没有作用， 因为事务将立即结束。

`_configuration_parameter_`

可设置的运行时参数的名字。可用的参数在Chapter 18和下面有文档。

`_value_`

参数的新值。值可以声明为字符串常量、标识符、数字，或者逗号分隔的上面这些东西的列表， 视特定变量而定。 对于特定参数是恰当的。可以写出 `DEFAULT` 把这些参数设置为它们的缺省值。（也就是说，如果在当前会话中没有 `SET` 执行， 已有的任何数值都会设置。）

除了在Chapter 18里面有文档记载的配置参数之外， 还有几个只能用 `SET` 命令设置，或者是有特殊的语法的参数。

`SCHEMA`

`SET SCHEMA '``_value_`'是 `SET search_path TO` `_value_` 的别名。 使用该语法只可指定一个模式。

`NAMES`

`SET NAMES` `_value_` 是 `SET client_encoding TO` `_value_` 的别名。

`SEED`

为随机数生成器(函数 `random`)设置内部的种子。 允许的值是介于 -1 和 1 之间的浮点数， 然后它会被乘以 2³¹-1。

也可以通过调用 `setseed` 函数来设置种子：

```
SELECT setseed(_value_);
```

`TIME ZONE`

`SET TIME ZONE` `_value_` 是 `SET timezone TO` `_value_` 的一个别名。语法 `SET TIME ZONE` 允许为时区设置特殊的语法。下面是有效值的例子：

`'PST8PDT'`

加州伯克利的时区。

`'Europe/Rome'`

意大利时区。

`-7`

UTC 以西 7 小时的时区(等效于 PDT)。正值为UTC以东。

`INTERVAL '-08:00' HOUR TO MINUTE`

UTC 以西 8 小时的时区(等效于 PST)。

`LOCAL``DEFAULT`

将时区设置为你的本地时区(也就是，服务器的 `timezone` 缺省值)。

参阅Section 8.5.3获取有关时区的更多细节。

注意

函数 `set_config` 提供了等效的功能。参阅Section 9.26。 同时，可以UPDATE这个 `pg_settings` 系统视图来执行 `SET` 的等价操作。

例子

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

把日期时间风格设置为传统的POSTGRES风格(日在月前)：

```
SET datestyle TO postgres, dmy;
```

把时区设置为加州伯克利：

```
SET TIME ZONE 'PST8PDT';
```

为意大利设置时区：

```
SET TIME ZONE 'Europe/Rome';
```

兼容性

`SET TIME ZONE` 扩展了在 SQL 标准里定义的语法。标准只允许有一个数字时区偏移， 而PostgreSQL还允许完整更灵活的时区声明。 所有其它的 `SET` 特性都是PostgreSQL扩展。

又见

RESET, SHOW

SET CONSTRAINTS

Name

SET CONSTRAINTS -- 设置当前事务的约束检查模式

Synopsis

```
SET CONSTRAINTS { ALL | _name_ [, ...] } { DEFERRED | IMMEDIATE }
```

描述

`SET CONSTRAINTS`设置当前事务里的约束检查的特性。 `IMMEDIATE` 约束是在每条语句后面进行检查。 `DEFERRED`约束一直到事务提交时才检查。 每个约束都有自己的`IMMEDIATE`或`DEFERRED`模式。

从创建的时候开始，一个约束总是给定为 `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, `NOT DEFERRABLE` 三个特性之一。 第三种总是`IMMEDIATE`, 并且不会受`SET CONSTRAINTS`影响。 头两种以指定的方式启动每个事务，但是他们的行为可以在事务里用`SET CONSTRAINTS`改变。

带着一个约束名列表的`SET CONSTRAINTS`改变这些约束的模式(都必须是可推迟的)。 每个约束名都可以是模式修饰的。如果没有指定任何模式名，那么使用当前模式搜索路径查找第一个匹配名。 `SET CONSTRAINTS ALL`改变所有可推迟约束的模式。

当`SET CONSTRAINTS`把一个约束从`DEFERRED`改成 `IMMEDIATE`的时候，新模式反作用式地起作用： 任何将在事务结束准备检查的数据修改都将在执行`SET CONSTRAINTS`的时候检查。 如果违反了任何约束， `SET CONSTRAINTS`都会失败(并且不会修改约束模式)。 因此， `SET CONSTRAINTS`可以用于强制在事务中某一点进行约束检查。

目前，仅`UNIQUE`, `PRIMARY KEY`, `REFERENCES` (外键), 和 `EXCLUDE`约束受该设置影响。 `NOT NULL` 和 `CHECK` 约束总在一行被插入或者修改时被检查（不在语句末）。 未声明`DEFERRABLE`的唯一性和排除性约束也立即检查。

被声明为"约束触发"的触发器的触发也是受该设置控制的， 他们在相关约束应被检查的相同时间触发。

注意

因为PostgreSQL不要求约束名称在一个模式内是独一无二的 （但每个表必须唯一），多于一个约束名匹配一个特定约束名是可能的。在这种情况下， `SET CONSTRAINTS` 将作用于所有的匹配。对于一个无模式限定的名称， 一旦一个或多个匹配在搜索路径下的相同模式中被找到，路径中晚出现的模式不会被搜索。

这个命令只在当前事务里修改约束的行为。因此，如果你在事务块之外 (`BEGIN` / `COMMIT` 对) 执行这个命令，它将没有任何作用。

兼容性

这条命令与 SQL 标准里定义的行为兼容，只不过，在PostgreSQL里， 它不适用于 `NOT NULL` 和 `CHECK` 约束。还有，PostgreSQL 立即检查不可推延的唯一性约束，不是像标准建议的那样在语句末检查。

SET ROLE

Name

SET ROLE -- 在当前会话中设置当前用户标识

Synopsis

```
SET [ SESSION | LOCAL ] ROLE _role_name_
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

描述

这条命令将当前会话的当前用户标识为 `_role_name_` 。 角色名可以写成表识符或者是字符串文本。在 `SET ROLE` 之后， SQL命令的权限检查就是针对这个用户了，就像当初用这个用户登录一样。

当前会话的用户必须是指定的 `_role_name_` 角色的成员。但超级用户可以选择任何角色。

`SESSION` 和 `LOCAL` 修饰词的作用和普通的SET命令一样。

`NONE` 和 `RESET` 形式重置当前用户标识为当前会话用户标识符。任何用户都可以执行这种形式。

注意

使用这条命令，它可能会增加一个用户的权限，也可能会限制一个用户的权限。 如果会话用户的角色有 `INHERITS` 属性，那么它自动拥有它能 `SET ROLE` 变成的角色的所有权限；在这种情况下， `SET ROLE` 实际上是删除了所有直接赋予会话用户的权限， 以及它的所属角色的权限，只剩下指定角色的权限。另一方面，如果会话用户的角色有 `NOINHERITS` 属性， `SET ROLE` 删除直接赋予会话用户的权限，而获取指定角色的权限。

实际上，如果一个超级用户 `SET ROLE` 为一个非超级用户，它会失去其超级用户权限。

`SET ROLE` 有和SET SESSION AUTHORIZATION类似的效果， 但是其中涉及的权限检查有区别。还有， `SET SESSION AUTHORIZATION` 决定其后有什么角色可以执行 `SET ROLE` 命令，而用 `SET ROLE` 并不修改稍后的 `SET ROLE` 可以设置的角色集。

`SET ROLE` 并不如同角色的ALTER ROLE设置所声明的那样处理会话变量；这只在登陆时发生。

`SET ROLE` 不能在 `SECURITY DEFINER` 函数内使用。

例子

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

兼容性

PostgreSQL允许标识符语法(`"rolename"`)，而SQL标准要求角色名字写成字符串文本。SQL并不允许在事务里面执行这条命令；PostgreSQL并未做此限制，因为没有理由限制。`SESSION` 和 `LOCAL` 修饰词是PostgreSQL 的扩展，还有 `RESET` 语法也一样。

又见

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

Name

SET SESSION AUTHORIZATION -- 为当前会话设置会话用户标识符和当前用户标识符

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION _user_name_
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

描述

这条命令把当前会话里的会话用户标识和当前用户标识都设置为 `_user_name_` 。这个用户名可以写成一个标识符或者一个字符串文本。 使用这个命令，可以临时变成一个非特权用户，稍后再切换回超级用户。

会话用户标识符一开始设置为(可能经过认证的)客户端提供的用户名。 当前用户标识符通常等于会话用户标识符，但是可能在 `SECURITY DEFINER` 的环境里或者类似的机制里临时改变。 也可以用SET ROLE来修改。 当前用户的身份和权限检查有关。

只有在最开始登录时使用的用户(已认证用户)具有超级用户权限，会话用户标识符才能改变成其它用户。否则，只能使用登录时使用认证用户，系统才接受该命令。

`SESSION` 和 `LOCAL` 修饰词和普通SET命令里的作用相同。

`DEFAULT` 和 `RESET` 形式重置会话和当前用户标识符为初始认证的用户名。 这些形式可以为任何用户执行。

注意

`SET SESSION AUTHORIZATION` 不能用在 `SECURITY DEFINER` 函数内。

例子

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许一些其它的表达式出现在文本 `_user_name_` 的位置上， 不过这个东西实际上并不重要。PostgreSQL允许标识符语法 (`"username"`)，而 SQL 不允许。SQL 不允许在一个事务的过程中用这条命令； PostgreSQL没有这个限制，因为没有什么理由不允许这样用。 `SESSION` 和 `LOCAL` 是PostgreSQL扩展， `RESET` 语法也是。

标准里头把执行这个命令所需的必要权限，交给具体的数据库实现自己去定义。

又见

SET ROLE

SET TRANSACTION

Name

SET TRANSACTION -- 设置当前事务的特性

Synopsis

```
SET TRANSACTION _transaction_mode_ [, ...]
SET TRANSACTION SNAPSHOT _snapshot_id_
SET SESSION CHARACTERISTICS AS TRANSACTION _transaction_mode_ [, ...]

 这里的 `_transaction_mode_`是下列之一:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

`SET TRANSACTION`命令为当前事务设置特性。它对后面的事务没有影响。`SET SESSION CHARACTERISTICS`为一个会话中随后的每个事务设置缺省的事务特性。 在单独的事务中，可以用`SET TRANSACTION`覆盖这些默认事务特性。

可用的事务特性是事务隔离级别、事务访问模式(读/写或者只读)和可推迟的模式。 另外，可以选择一个快照，只为当前事务而不是作为会话缺省。

事务的隔离级别决定一个事务在有其它事务并发运行时它能够看到什么数据

`READ COMMITTED`

一条语句只能看到在它开始之前的数据。这是缺省。

`REPEATABLE READ`

当前事务中的所有语句只能看到在这次事务第一条查询或者修改数据的语句执行之前已经提交的数据。

`SERIALIZABLE`

当前事务中的所有语句只能看到在这次事务第一条查询或者修改数据的语句执行之前已经提交的数据。 如果当前并发的可串行化事务的读或写模式中任意一个事务（一次一个）执行创建不可能发生的情况， 那么其中的一个事务将会带有`serialization_failure`错误退出。

SQL 标准还定义了另外一个级别，`READ UNCOMMITTED`。 在PostgreSQL里 `READ UNCOMMITTED` 被当作 `READ COMMITTED`。

事务隔离级别在事务中第一个查询或数据修改语句(`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, 或 `COPY`)执行之后就不能再次设置。 参阅Chapter 13 获取有关事务隔离级别和并发性控制的更多信息。

事务访问模式决定事务是读/写还是只读。读/写是缺省。如果一个事务是只读， 而且写入的表不是临时表，那么下面的 SQL 命令是不允许的：`INSERT`, `UPDATE`, `DELETE`, `COPY FROM`； 而所有的 `CREATE`, `ALTER`, `DROP`; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; `EXPLAIN ANALYZE`, `EXECUTE` 都不允许。这是一个高层次的只读概念， 它并不阻止所有对磁盘的写入。

`DEFERRABLE` 事务属性没什么影响，除非事务也是 `SERIALIZABLE` 和 `READ ONLY`。当为一个事务选择所有这三个属性时， 事务可能会在第一次请求它的快照时堵塞，之后运行时可以不需要正常的 `SERIALIZABLE` 事务的开销并且不会有任何导致序列化失败或被取消的风险。 这个模式很适合于长时间运行的报告或备份。

`SET TRANSACTION SNAPSHOT` 命令允许一个新的事务运行的快照 与一个现存事务的相同。该已有的事务必须已经用 `pg_export_snapshot` 函数 （参阅Section 9.26.5）输出了它的快照。 这个函数返回一个快照标识符，这个标识符必须给 `SET TRANSACTION SNAPSHOT` 以指出要输出哪个快照。该标识符在这个命令中必须写作字符串，例如 `'000003A1-1'`。 `SET TRANSACTION SNAPSHOT` 只能在该事务第一次查询或数据修改语句执行之前设置， (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, `COPY`)。另外，事务必须已经设置为 `SERIALIZABLE` 或 `REPEATABLE READ` 隔离级别（否则，快照会被立即丢弃，因为 `READ COMMITTED` 模式为每个命令接受一个新的快照）。如果导入事务使用 `SERIALIZABLE` 隔离级别，那么输出快照的事务必须也使用该隔离级别。 还有，非只读的序列化事务不能从一个只读的事务中导入一个快照。

注意

如果执行 `SET TRANSACTION` 之前没有执行 `START TRANSACTION` 或 `BEGIN`，那么它会显得没有效果一样，因为事务将立即结束。

可以用在 `BEGIN` 或 `START TRANSACTION` 里面声明所需要的 `_transaction_modes_` 的方法来避免使用 `SET TRANSACTION`。但是这个选项不适用于 `SET TRANSACTION SNAPSHOT`。

会话的缺省事务模式也可以通过设置配置参数 default_transaction_isolation, default_transaction_read_only, 和 default_transaction_deferrable的方法来设置。 (实际上 `SET SESSION CHARACTERISTICS` 只是一个用 `SET` 来设置这些参数的另一种等效的方法)。这就意味着缺省值可以通过 `ALTER DATABASE` 或在配置文件里等方法设置。参考Chapter 18获取更多信息。

例子

开启一个与已经存在的事务具有相同快照的新事务，首先从现存事务中输出快照。 这将返回快照的标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
--------------------
 000003A1-1
(1 row)
```

然后在新打开的事务的开始的 `SET TRANSACTION SNAPSHOT` 命令中给出快照标识符：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '000003A1-1';
```

兼容性

这些命令在SQL标准中定义了，除了 `DEFERRABLE` 事务模式和 `SET TRANSACTION SNAPSHOT` 格式是PostgreSQL 的扩展。

标准里的缺省事务隔离级别是 `SERIALIZABLE` ；但在PostgreSQL里，缺省隔离级别是 `READ COMMITTED`， 但是你可以用上面描述的方法修改它。

在 SQL 标准里还有另外一种事务特性可以用这些命令设置：诊断范围的大小。 这个概念只用于嵌入式 SQL，因此没有在PostgreSQL服务器里实现。

SQL 标准要求在相连的 `_transaction_modes_` 之间使用逗号，但是因为历史原因，PostgreSQL允许省略这个逗号。

SHOW

Name

SHOW -- 显示运行时参数的值

Synopsis

```
SHOW _name_
SHOW ALL
```

描述

`SHOW` 将显示当前运行时参数的数值。这些变量可以通过 `SET` 语句、编辑 `postgresql.conf` 文件、 `PGOPTIONS` 环境变量(在使用基于libpq的应用程序的时候)、 启动 `postgres` 服务器的命令行参数来设置。参阅Chapter 18获取细节。

参数

`_name_`

运行时参数的名称。可用的参数在Chapter 18以及SET 手册页里面有文档。 另外， 还有一些参数可以显示， 但是不能设置：

`SERVER_VERSION`

显示服务器的版本号。

`SERVER_ENCODING`

显示服务器端的字符集编码。目前， 这个参数只能显示但不能设置， 因为编码是在创建数据库的时候决定的。

`LC_COLLATE`

显示数据库的排序规则区域设置(文本顺序)。目前， 这个参数只能显示但不能设置， 因为它是在数据库创建的时候设置的。

`LC_CTYPE`

显示数据库字符集分类的区域设置。目前， 这个参数只能显示但不能设置， 因为它是在数据库创建的时候设置的。

`IS_SUPERUSER`

如果当前角色拥有超级用户权限，则为真。

`ALL`

显示所有配置参数值以及其描述。

注意

函数 `current_setting` 生成相同的输出; 参阅Section 9.26。还有， `pg_settings` 系统视图生成同样的信息。

例子

显示参数 `DateStyle` 的当前设置：

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

显示参数 `geqo` 的当前设置：

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

显示所有设置：

```
SHOW ALL;
            name         | setting |                description
-------------------------+---------+-------------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

兼容性

`SHOW` 命令是PostgreSQL扩展。

又见

SET, RESET

START TRANSACTION

Name

START TRANSACTION -- 开始一个事务块

Synopsis

```
START TRANSACTION [ _transaction_mode_ [, ...] ]

这里的 `_transaction_mode_`是下列之一：

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

这条命令开始一个新的事务块。如果声明了隔离级别、读写模式或可推迟模式， 那么新事务就使用这些特性，如同执行了SET TRANSACTION一样。 它和BEGIN命令等价。

参数

参阅SET TRANSACTION获取有关这个语句参数含义的信息。

兼容性

在标准里，没必要声明 `START TRANSACTION` 来开始一个事务块： 任何 SQL 语句都隐含地开始一个事务块。PostgreSQL 的行为可以认为是隐含地在每条没有跟在 `START TRANSACTION` 或 `BEGIN` 后面的命令自动发出一条 `COMMIT`， 因此这个行为常被称作"自动提交"。 其它关系数据库系统可能也提供自动提交这一方便的特性。

`DEFERRABLE` `_transaction_mode_` 是一个PostgreSQL语言扩展。

SQL 标准要求在相连的 `_transaction_modes_` 之间有逗号，但是出于历史原因，PostgreSQL 允许省略这个逗号。

又见SET TRANSACTION的兼容性小节。

又见

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

Name

TRUNCATE -- 清空一个或一组表

Synopsis

```
TRUNCATE [ TABLE ] [ ONLY ] _name_ [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

描述

`TRUNCATE` 快速地从一堆表中删除所有行。它和在每个表上进行无条件的 `DELETE` 有同样的效果，不过因为它不做表扫描，因而快得多。另外， 它快速回收磁盘空间，而不是请求一个随后的 `VACUUM` 操作。 在大表上最有用。

参数

`_name_`

要清空的表名字(可以有模式修饰)。如果在表名前声明了 `ONLY` ， 那么只有那个表会被清空。如果没有声明 `ONLY` ， 这个表以及其所有子表 （若有）会被清空。可选地,可以在表名后声明 `*` 以明确指定包括子表。

`RESTART IDENTITY`

截断表中的序列字段的序列值将被重置。

`CONTINUE IDENTITY`

不要改变序列的值。此为缺省。

`CASCADE`

级联清空所有在该表上有外键引用的表，或者由于 `CASCADE` 而被添加到组中的表。

`RESTRICT`

如果有外键引用数据并且引用表没有在命令行中列出，则拒绝清空。这是缺省。

注意

你必须对表有 `TRUNCATE` 权限以清空它。

`TRUNCATE` 在在操作的每个表上请求一个 `ACCESS EXCLUSIVE` 锁， 这种锁会阻塞表上的所有其他并发操作。当声明了 `RESTART IDENTITY` 时， 任何被重启的序列也会被排他锁锁住。如果需要对一个表并发访问， 那么应该使用 `DELETE` 命令。

如果从其它表有到某个表的外键引用，那么就不能对该表使用 `TRUNCATE` ， 除非这些表在同一个命令中也被清空。在这种情况下检查有效性要求进行表扫描， 而 `TRUNCATE` 并不做这样的事情。 `CASCADE` 选项可以用于级联包含所有依赖表，但是使用此选项必须十分小心，否则可能丢失原本不想丢失的数据。

`TRUNCATE` 不会触发任何在该表上的 `ON DELETE` 触发器。 但它会触发 `ON TRUNCATE` 触发器。如果任何表上定义有 `ON TRUNCATE` 触发器，则所有的 `BEFORE TRUNCATE` 触发器会在任何截断发生之前触发，而所有 `AFTER TRUNCATE` 触发器会在最后一个截断执行后和任何序列重设后被触发。该触发器会以表执行的顺序触发 （首先是命令中列出的，然后是任何基于级联添加的）。

Warning
`TRUNCATE` 不是mvcc 安全的操作（参考mvcc的文档）。在截断之后，所有并发的事务看到的该表都是空的，即使他们在截断发生前使用快照。 这将是一个针对截断发生之前没有访问截断表的事务的问题，在事务中对截断表的任何访问至少产生一个 `ACCESS SHARE` 锁，这将阻塞 `TRUNCATE` 直到事务完成。 所以截断不会导致在相同表上连续查询时表内容上的明显不一致， 但会导致截断表和数据库中其他表在内容上可见的不一致性。

`TRUNCATE` 在事务中断表数据是安全可的：只要开启的事物不提交，则事物会安全回滚。

当声明了 `RESTART IDENTITY` 时，隐含的 `ALTER SEQUENCE RESTART` 操作也会事务性的完成；也就是，如果包围的事务没有提交，它们将回滚。 这不同于 `ALTER SEQUENCE RESTART` 的正常行为。要知道， 如果在序列重启之后与事务回滚之前执行了额外的序列操作， 那么这些操作的影响也会回滚，但是它们在 `currval()` 上的影响不会回滚； 也就是，在事务之后， `currval()` 将继续反映在失败事务中获得的最后序列值， 即使序列本身可能与其不再一致。这类似于 `currval()` 在一个失败事务后的普通行为。

例子

清空 `bigtable` 和 `fattable` 表：

```
TRUNCATE bigtable, fattable;
```

同样地，也重置任何相关的序列生成器:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

清空 `othertable` 表，并且级联清空所有通过外键约束引用 `othertable` 的表：

```
TRUNCATE othertable CASCADE;
```

兼容性

SQL:2008标准包括一个有语法 `TRUNCATE TABLE` `_tablename_` 的 `TRUNCATE` 命令。 `CONTINUE IDENTITY` / `RESTART IDENTITY` 也出现在那个标准中，但是有轻微的不同和相关的含义。该命令的一些并发行为通过标准定义实现的， 所以，如果必要，应该考虑上面的说明并与其他实现比较。

UNLISTEN

Name

UNLISTEN -- 停止监听通知信息

Synopsis

```
UNLISTEN { _channel_ | * }
```

描述

`UNLISTEN` 用于删除一个现有的已注册 `NOTIFY` 事件。 `UNLISTEN` 取消当前PostgreSQL 会话中所有对通知通道名为 `_channel_` 的监听。 特殊的条件通配符 `*` 取消对当前会话的所有监听。

NOTIFY包含一些对 `LISTEN` 和 `NOTIFY` 的更广泛的讨论。

参数

`_channel_`

通知通道名称(任意标识符)。

`*`

该会话当前的所有监听都将被清除。

注意

即使取消一个你没有监听的事件，后端也不会警告或报错。

每个会话在退出时都会自动执行 `UNLISTEN *` 。

一个已经执行了 `UNLISTEN` 的事务不能为二阶段提交做准备。

例子

注册一个：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

一旦执行了 `UNLISTEN`，以后的 `NOTIFY` 信息都将被忽略：

```
UNLISTEN virtual;
NOTIFY virtual;
-- 收不到 NOTIFY 事件
```

兼容性

SQL 标准里没有 `UNLISTEN` 命令。

又见

LISTEN, NOTIFY

UPDATE

Name

UPDATE -- 更新一个表中的行

Synopsis

```
[ WITH [ RECURSIVE ] _with_query_ [, ...] ]
UPDATE [ ONLY ] _table_name_ [ * ] [ [ AS ] _alias_ ]
    SET { _column_name_ = { _expression_ | DEFAULT } |
          ( _column_name_ [, ...] ) = ( { _expression_ | DEFAULT } [, ...] ) } [, ...]
    [ FROM _from_list_ ]
    [ WHERE _condition_ | WHERE CURRENT OF _cursor_name_ ]
    [ RETURNING * | _output_expression_ [ [ AS ] _output_name_ ] [, ...] ]
```

描述

`UPDATE` 改变满足条件的所有行中指定的字段值。 只在 `SET` 子句中出现需要修改的行，没有出现的其他字段保持它们原来的数值。

使用同一数据库里其它表的信息来更新一个表有两种方法：使用子查询， 或者在 `FROM` 子句里声明另外一个表。哪个方法更好取决于具体的环境。

可选的 `RETURNING` 子句将导致 `UPDATE` 基于每个被更新的行计算返回值。 任何使用表的字段的表达式或 `FROM` 中使用的其他表的字段都可以用于计算。 计算的时候使用刚刚被更新过的字段新值。 `RETURNING` 列表的语法与 `SELECT` 的输出列表相同。

你必须对表或至少在列出的要被更新的字段上有 `UPDATE` 权限， 同样在 `_expressions_` 或 `_condition_` 条件里提到的任何字段上也要有 `SELECT` 权限。

参数

`_with_query_`

`WITH` 子句允许声明一个或多个可以在 `UPDATE` 查询中通过名字引用的子查询。参阅Section 7.8 和 SELECT获取详细信息。

`_table_name_`

要更新的表的名称(可以有模式修饰)。如果在表名前声明了 `ONLY` ， 那么只在该表中更新匹配的行。如果没有声明 `ONLY` ，那么该表的任何继承表内的匹配行也要更新。 可选的， 可以在表名后面声明 `*` 以明确表明包含后代表。

`_alias_`

目标表的别名。如果指定了别名，那么它将完全遮盖表的本名。例如，给定 `UPDATE foo AS f` 之后，剩余的 `UPDATE` 语句必须用 `f` 而不是 `foo` 引用这个表。

`_column_name_`

表 `_table_name_` 中的字段名。必要时， 字段名可以用子域名或者数组下标修饰。不要在指定字段名的时候加上表名字。 比如 `UPDATE tab SET tab.col = 1` 就是错误的。

`_expression_`

给字段赋值的表达式。表达式可以使用这个或其它字段更新前的旧值。

`DEFAULT`

把字段设置为它的缺省值，如果没有缺省表达式，那么就是 NULL。

`_from_list_`

一个表表达式的列表，允许来自其它表中的字段出现在 `WHERE` 条件里和更新表达式中。 这个类似于可以在一个 `SELECT` 语句的*FROM 子句* 子句里声明表列表。请注意目标表绝对不能出现在 `_from_list_` 里， 除非你是在使用一个自连接(此时它必须以 `_from_list_` 的别名出现)。

`_condition_`

一个返回 `boolean` 结果的表达式。只有这个表达式返回 `true` 的行才会被更新。

`_cursor_name_`

在 `WHERE CURRENT OF` 条件下使用的游标的名称。要更新的行是最近从该游标中抓去的行。 该游标必须是一个 `UPDATE` 目标表中的非分组查询。请注意 `WHERE CURRENT OF` 不能与布尔条件一起声明。参阅DECLARE获取更多关于通过 `WHERE CURRENT OF` 使用游标的信息。

`_output_expression_`

在所有需要更新的行都被更新之后， `UPDATE` 命令用于计算返回值的表达式。 这个表达式可以使用任何 `_table_name_` 命名的表以及 `FROM` 中列出的表的字段。写上 `*` 表示返回所有字段。

`_output_name_`

字段的返回名称。

Outputs

成功完成后， `UPDATE` 返回形如

```
UPDATE _count_
```

的命令标签。 `_count_` 是更新的行数， 包括没有改变值的匹配行。请注意当更新受到 `BEFORE UPDATE` 触发器的抑制时， 这个数字可能小于匹配 `_condition_` 的行数。 如果 `_count_` 为 0，则没有行被这个查询更新 (这个不认为是错误)。

如果 `UPDATE` 包含 `RETURNING` 子句，那么返回的结果将类似于包含在 `RETURNING` 列表中定义的字段和值的 `SELECT` 语句， 只不过返回结果是基于被更新的行而已。

注意

在出现 `FROM` 子句的时候，实际上发生的事情是目标表和 `_from_list_` 里提到的表连接在一起，并且每个连接输出行都代表一个目标表的更新操作。 在使用 `FROM` 的时候，你应该保证连接为每个需要修改的行最多生成一个输出行。 换句话说，一个目标行不应该和超过一行来自其它表的数据行连接。如果它连接了多于一个行， 那么连接行里面将会只有一行用于更新目标行，但是究竟使用哪行是很难预期的事情。

因为这个不确定性，只在子查询里面引用其它表是安全的，尽管通常更难读并且比使用连接也更慢些。

例子

把表 `films` 里的字段 `kind` 里的词 `Drama` 用 `Dramatic` 代替：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

调整表 `weather` 中的某行的温度并把该行的降水量设置为缺省值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

做同样的事情并返回更新后的条目：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

使用另一种字段列表语法来做同样的事情：

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

增加负责 Acme 公司客户的销售员的销售计数，使用 `FROM` 子句语法：

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

使用 `WHERE` 子句里的子查询执行同样的操作：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

试图带着库存量插入一个新的库存项。如果该项存在，则更新现有项的库存数。 要做这件事情而又不使整个事务失效，使用保留点。

```
BEGIN;
-- 其它操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假设上面因为一个唯一键字违例而失效,
-- 因此现在发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 继续其它操作，最后
COMMIT;
```

更改表 `films` 的 `kind` 列， 在游标 `c_films` 目前定位的行上：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

兼容性

这条命令遵循SQL标准。只是 `FROM` 和 `RETURNING` 子句是PostgreSQL扩展，就像和 `UPDATE` 一起使用 `WITH` 。

标准的字段列表语法允许从行值表达式指定字段列表，比如一个子查询：

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
    (SELECT last_name, first_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

这个功能目前尚未实现：源必须是一个独立的表达式。

有些其它数据库系统提供一个 `FROM` 选项，在这个选项下， 认为目标表会再次在 `FROM` 里列出。这不是PostgreSQL 解析 `FROM` 的方式。移植使用这类扩展的应用时要注意。

VACUUM

Name

VACUUM -- 垃圾收集以及可选地分析一个数据库

Synopsis

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ _table_name_ [ (_column_na
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ _table_name_ ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ _table_name_ [ (_column_name_ [, ...] )
```

描述

`VACUUM` 回收死行占据的存储空间。在一般的PostgreSQL 操作里，那些已经 DELETE 的行或者被 UPDATE 过后过时的行并没有从它们所属的表中物理删除； 在完成 `VACUUM` 之前它们仍然存在。因此有必要周期地运行 `VACUUM` ， 特别是在经常更新的表上。

如果没有参数， `VACUUM` 处理当前用户有权限vacuum的当前数据库里的每个表， 如果有参数， `VACUUM` 只处理那个表。

`VACUUM ANALYZE` 先执行一个 `VACUUM` 然后是给每个选定的表执行一个 `ANALYZE` 。对于日常维护脚本而言， 这是一个很方便的组合。参阅ANALYZE获取更多有关其处理的细节。

简单的 `VACUUM` (没有 `FULL`)只是简单地回收空间并且令其可以再次使用。 这种形式的命令可以和对表的普通读写并发操作，因为没有请求排他锁。然而， 额外的空间并不返回到操作系统（在大多数情况下）；仅保持在相同的表中可重用。 `VACUUM FULL` 将表的全部内容重写到一个没有任何多余空间的新磁盘文件中， 允许未使用的空间返回到操作系统中。这种形式要慢许多并且在处理的时候需要在表上施加一个排它锁。

当选项列表被括号括起来时，该选项可以任意顺序来写。没有圆括号，选项必须按以上显示的顺序指定。 加圆括号的语法在PostgreSQL 9.0中添加；不加括号的语法已经废弃了。

参数

`FULL`

选择"完全"清理，这样可以恢复更多的空间，但是花的时间更多并且在表上施加了排它锁。该方法也需要额外的磁盘空间，因为它写了一个表的新拷贝并且不释放旧的拷贝，直到操作完成。 通常这应该只用于当一个大量的空间需要在这个表中回收时。

`FREEZE`

选择激进的行"冻结"。指定 `FREEZE` 相当于执行 `VACUUM` 时将vacuum_freeze_min_age参数设为零。

`VERBOSE`

为每个表打印一份详细的清理工作报告。

`ANALYZE`

更新用于优化器的统计信息，以决定执行查询的最有效方法。

`_table_name_`

要清理的表的名称(可以有模式修饰)。缺省时是当前数据库中的所有表。

`_column_name_`

要分析的具体的字段名称。缺省是所有字段。若指定一个字段列表，就暗含 `ANALYZE` 。

输出

如果指定了 `VERBOSE` ，那么 `VACUUM` 将发出处理过程中的信息， 以表明当前正在处理哪个表。各种有关这些表的统计也会打印出来。

注意

要清空一个表，一个人必须通常是表的所有者或是一个超级用户。然而， 数据库所有者允许清空他们的数据库中的所有表，除了共享目录。 （对共享目录的限制意味着一个真正的数据库范围的 `VACUUM` 仅能由超级用户执行。） `VACUUM` 将跳过调用用户没有权限清空的任何表。

`VACUUM` 不能在事务块内执行。

对于有GIN索引的表， `VACUUM` （以任何形式） 也完成任何挂起索引插入内容，通过移动挂起索引条目到主GIN索引结构中的相应位置。 参阅Section 57.3.1获取详细信息。

建议生产数据库经常清理(至少每晚一次)，以保证不断地删除死行。尤其是在增删了大量记录之后， 对受影响的表执行 `VACUUM ANALYZE` 命令是一个很好的习惯。 这样做将更新系统目录为最近的更改，并且允许PostgreSQL 查询优化器在规划用户查询时有更好的选择。

不建议日常使用 `FULL` 选项，但是可以在特殊情况下使用。 一个例子就是在你删除或更新了一个表的大部分行之后， 希望从物理上缩小该表以减少磁盘空间占用并且允许更快的表扫描。 `VACUUM FULL` 通常要比单纯的 `VACUUM` 收缩更多的表尺寸。

`VACUUM` 导致 I/O 流量增加，可能会导致其它活动会话的性能恶劣。因此， 有时候会建议使用基于开销的 vacuum 延迟特性。 参阅Section 18.4.4获取细节。

PostgreSQL包含一个"autovacuum"设施， 它可以自动进行日常的 vacuum 维护。关于手动和自动清理的更多细节， 参见Section 23.1。

例子

下面是一个在蜕变(regression)数据库里某个表上执行 `VACUUM` 的一个例子：

```
regression=# VACUUM (VERBOSE, ANALYZE) onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_stringu1" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

兼容性

SQL 标准里没有 `VACUUM` 语句。

又见

vacuumdb, Section 18.4.4, Section 23.1.6

VALUES

Name

VALUES -- 计算一个或一组行

Synopsis

```
VALUES ( _expression_ [, ...] ) [, ...]
    [ ORDER BY _sort_expression_ [ ASC | DESC | USING _operator_ ] [, ...] ]
    [ LIMIT { _count_ | ALL } ]
    [ OFFSET _start_ [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ _count_ ] { ROW | ROWS } ONLY ]
```

描述

`VALUES` 根据给定的值表达式计算一个或一组行的值。 它通常用于在一个较大的命令内生成一个"常数表"，但是它也可以用于自身。

如果指定了多行，那么每一行都必须拥有相同的元素个数。 结果表字段的数据类型将根据表达式的明确或隐含的数据类型确定， 使用的规则与 `UNION` (Section 10.5)相同。

在大命令里面， `VALUES` 允许出现在任何 `SELECT` 可以出现的地方。 因为它在语法上非常类似于 `SELECT` ， 可以在 `VALUES` 命令中使用 `ORDER BY` ， `LIMIT` (或相等的 `FETCH FIRST`), 和 `OFFSET` 子句。

参数

`_expression_`

用于计算或插入结果表指定地点的常量或者表达式。在一个出现在 `INSERT` 顶层的 `VALUES` 列表中， `_expression_` 可以被 `DEFAULT` 替换以表示插入目的字段的缺省值。除此以外，当 `VALUES` 出现在其他场合的时候是不能使用 `DEFAULT` 的。

`_sort_expression_`

一个表示如何排序结果行的表达式或者整数常量。这个表达式可以按照 `column1` ， `column2` 等等引用 `VALUES` 的结果列。更多细节参见 *ORDER BY 子句*。

`_operator_`

一个排序操作符。更多细节参见*ORDER BY 子句*。

`_count_`

返回的最大行数。更多细节参见 *LIMIT 子句*。

`_start_`

在开始返回行之前跳过的行数。更多细节参见*LIMIT 子句*。

注意

应当避免使用 `VALUES` 返回数量非常大的结果行，否则可能会遭遇内存耗尽或者性能低下。 出现在 `INSERT` 中的 `VALUES` 是一个特殊情况， 因为目标字段类型可以从 `INSERT` 的目标表获知，并不需要通过扫描 `VALUES` 列表来推测，所以在此情况下可以处理非常大的结果行。

例子

一个光秃秃的 `VALUES` 命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个三行两列的表，它在效果上相当于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

通常， `VALUES` 会用于一个大 SQL 命令中，最常见的是用于 `INSERT` 命令：

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在用于 `INSERT` 时， `VALUES` 列表中的项可以使用 `DEFAULT` 来表示使用字段的缺省值：

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

`VALUES` 还可以用于子 `SELECT` 可以应用的场合。 比如在一个 `FROM` 子句中：

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

当 `VALUES` 用于 `FROM` 子句中的时候，必须使用 `AS` 子句， 这与用于 `SELECT` 时一样。并不要求 `AS` 子句为每个字段都指定一个别名， 但是这样做是一个好习惯。PostgreSQL中 `VALUES` 缺省的字段名是 `column1` ，`column2` 等等， 但这些缺省的名字并不一定与其他数据库系统相同。

当 `VALUES` 用于 `INSERT` 的时候，所有的值都将按照目标字段自动做类型转换。 但是在其他场合就可能必须明确指定恰当的数据类型。如果所有的项都是引号包围的字面常量， 强制指定第一个类型就可以确定所有的类型：

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

> **Tip:** 对于简单的 `IN` 测试，更好的方法是依赖于 `IN` 的标量列表形式， 而不是使用上述 `VALUES` 查询。标量列表的方法写的更少，而且通常也更有效。

兼容性

`VALUES` 符合 SQL 标准，此外， `LIMIT` 和 `OFFSET` 是PostgreSQL扩展； 又见SELECT。

又见

INSERT, SELECT

II. PostgreSQL 客户端应用程序

这部分包含PostgreSQL客户端应用程序和工具的参考信息。 这里的命令并非全部是通用工具，有些可能需要特殊的权限。 这些应用的一般性特点是它们可以在任何主机上运行，与数据库服务器所处的位置无关。

当在命令行指定时，用户和数据库名保留大小写，空格或特殊字符可能需要引起来。 表名和其他标识符不保留大小写（除了那些记录）并且需要引起来。

Table of Contents

clusterdb

Name

clusterdb -- cluster a PostgreSQL database

Synopsis

`clusterdb` [`_connection-option_` ...] [`--verbose` | `-v`] [`--table` | `-t` `_table_`] ...
[`_dbname_`]

`clusterdb` [`_connection-option_` ...] [`--verbose` | `-v`] `--all` | `-a`

Description

clusterdb is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

clusterdb is a wrapper around the SQL command CLUSTER. There is no effective difference between clustering databases via this utility and via other methods for accessing the server.

Options

clusterdb accepts the following command-line arguments:

`` -a``--all ``

Cluster all databases.

`[-d]` `_dbname_` `` [--dbname=]``_dbname_ ``

Specifies the name of the database to be clustered. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE` . If that is not set, the user name specified for the connection is used.

`-e` `--echo`

Echo the commands that clusterdb generates and sends to the server.

`-q` `--quiet`

Do not display progress messages.

`-t` `_table_` `--table=``_table_`

Cluster `_table_` only. Multiple tables can be clustered by writing multiple `-t` switches.

`-v` `--verbose`

Print detailed information during processing.

`-V` `--version`

Print the clusterdb version and exit.

`-?` `--help`

Show help about clusterdb command line arguments, and exit.

clusterdb also accepts the following command-line arguments for connection parameters:

`-h` `_host_` `--host=``_host_`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` `_port_` `--port=``_port_`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` `_username_` `--username=``_username_`

User name to connect as.

`-w` `--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W` `--password`

Force clusterdb to prompt for a password before connecting to a database.

This option is never essential, since clusterdb will automatically prompt for a password if the server demands password authentication. However, clusterdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=``_dbname_`

Specifies the name of the database to connect to discover what other databases should be clustered. If not specified, the `postgres` database will be used, and if that does not exist, `template1` will be used.

Environment

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see CLUSTER and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To cluster the database `test` :

```
<samp class="literal">$</samp> <kbd class="literal">clusterdb test</kbd>
```

To cluster a single table `foo` in a database named `xyzzy` :

```
<samp class="literal">$</samp> <kbd class="literal">clusterdb --table foo xyzzy</kbd>
```

See Also

CLUSTER

createdb

Name

createdb -- 创建一个新 PostgreSQL 数据库

Synopsis

`createdb` [`_connection-option_` ...] [`_option_` ...] [`_dbname_` [`_description_`]]

描述

createdb 创建一个新 PostgreSQL 数据库。

通常，执行这个命令的数据库用户将成为新数据库的所有者。 不过，如果拥有合适的权限，那么也可以通过 `-O` 指定其他用户。

createdb 是一个 SQL 命令 CREATE DATABASE的封装。 因此，用哪种方法创建的数据库都一样。

选项

createdb 接受下列命令行参数：

`_dbname_`

要创建的数据库名。该名称应该在本节点的所有PostgreSQL 数据库中是唯一的。 缺省创建的数据库名称是与当前系统用户同名。

`_description_`

指定与新创建的数据库相关的注释。

`-D` `_tablespace_``--tablespace=``_tablespace_`

指定数据库默认表空间。（这个名称被当作一个用双引号括起来的标识符处理。）

`-e` `--echo`

回显 createdb 生成并发送到服务端的命令。

`-E` `_encoding_` `--encoding=``_encoding_`

指定在此数据库中使用的字符编码方案。 PostgreSQL 服务器支持的字符集在 Section 22.3.1 里列出。

`-l` `_locale_` `--locale=``_locale_`

指定在此数据库中使用的语言环境。 这相当于同时指定 `--lc-collate` 和 `--lc-ctype` 选项。

`--lc-collate=``_locale_`

指定数据库的LC_COLLATE设置。

`--lc-ctype=``_locale_`

指定数据库的LC_CTYPE设置。

`-O` `_owner_` `--owner=``_owner_`

指定将拥有新数据库的用户。 （这个名称被当作一个用双引号括起来的标识符处理。）

`-T` `_template_` `--template=``_template_`

指定创建此数据库的模板数据库。 （这个名称被当作一个用双引号括起来的标识符处理。）

`-V` `--version`

输出createdb命令的版本信息，然后退出。

`-?` `--help`

显示createdb命令的帮助信息，然后退出。

选项 `-D`，`-l`，`-E`，`-O`，和 `-T` 对应底层的SQL命令 CREATE DATABASE的选项； 更多信息可以参考该命令的手册页。

createdb还接受以下命令行选项用于连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

进行联接的用户名。

`-w` `--no-password`

永远不提示输入密码。 如果服务器要求密码验证和密码 通过其他方式如 `.pgpass` 文件（验证）不可用，则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制createdb联接到数据库之前提示输入密码。

这个选项不是必须的，如果服务器要求认证密码 createdb会自动提示需输入密码。 然而，createdb会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

`--maintenance-db=``_dbname_`

指定要创建新数据库时联接的数据库名称。 如果没有指定，将使用 `postgres` 数据库； 如果不存在（或是正在创建的新数据库的名称）， 将使用 `template1` 数据库。

环境变量

`PGDATABASE`

如果设置了，那么就是要创建的新数据库的名称，除非在命令行上进行覆盖。

`PGHOST` `PGPORT` `PGUSER`

缺省的联接参数。 如果没有在命令行上和 `PGDATABASE` 中声明数据库名称的话， `PGUSER` 还决定了要创建的数据库名称。 .

此实用工具，像大多其他的PostgreSQL实用工具， 还使用 libpq支持的环境变量（见 Section 31.14）。

诊断

如果有问题,将会显示后端错误信息。 参阅CREATE DATABASE 和psql获取可能的问题和错误消息的描述。 数据库服务端必须运行在目标主机。 此外，前端库libpq中的所有缺省连接设置和环境变量都将适用。

示例

用缺省数据库服务器创建一个 `demo` 数据库：

```
<samp class="literal">$</samp> <kbd class="literal">createdb demo</kbd>
```

在主机 `eden` 上创建 `demo` 数据库，端口是5000， 使用 `LATIN1` 编码方案， 并且回显执行的命令：

```
<samp class="literal">$</samp> <kbd class="literal">createdb -p 5000 -h eden -E LATIN1 -e
<samp class="literal">CREATE DATABASE demo ENCODING 'LATIN1';</samp>
```

另请参阅

dropdb, CREATE DATABASE

createlang

Name

createlang -- 安装一个PostgreSQL过程语言

Synopsis

`createlang` [`_connection-option_` ...] `_langname_` [`_dbname_`]

`createlang` [`_connection-option_` ...] `--list` | `-l` [`_dbname_`]

描述

createlang是一个用于向 PostgreSQL数据库中增加编程语言的工具。

createlang仅是对SQL命令 CREATE EXTENSION的封装。

Caution

createlang是一个废弃了的可能在未来 PostgreSQL版本中删除的命令。 建议直接使用 `CREATE EXTENSION` 命令。

选项

createlang接受下列命令行参数：

`_langname_`

指定要安装的程序语言的名称。 （名称需小写。）

`[-d]` `_dbname_``[--dbname=]``_dbname_`

指定向哪个数据库增加该语言。缺省使用和当前系统用户同名的数据库。

`-e` `--echo`

回显所执行的SQL命令。

`-l` `--list`

显示在目标数据库中已经安装的语言的列表。

`-V` `--version`

输出createlang命令的版本信息，然后退出。

`-?` `--help`

显示createlang命令的帮助信息，然后退出。

createlang还接受以下命令行选项用于联接参数：

`-h` `_host_` ` --host=``_host_ `

指定运行服务器的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` ` --port=``_port_ `

指定服务器侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` ` --username=``_username_ `

进行联接的用户名。

`-w` `--no-password`

永远不提示输入密码。 如果服务器要求密码验证和密码 （并且）通过其他方式如 `.pgpass` 文件（验证）不可用， 则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制createlang联接到数据库之前提示输入密码。

这个选项不是必须的，如果服务器要求认证密码 createlang会自动提示需输入密码。 然而，createlang会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

环境变量

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

缺省的联接参数

此实用工具，像大多其他的PostgreSQL实用工具， 还使用libpq支持的环境变量 (见 Section 31.14)。

诊断

大多数错误信息都是自解释的。如果不是，使用 `--echo` 选项运行createlang然后查看相应的SQL命令的详细信息。 此外，前端库libpq中的所有缺省连接设置和环境变量都将适用。

注意

使用droplang删除一种语言。

示例

把 `pltcl` 语言安装到数据库 `template1` 里：

```
<samp class="literal">$</samp> <kbd class="literal">createlang pltcl template1</kbd>
```

注意，安装到 `template1` 中的语言将自动安装到随后创建的其他数据库中。

另请参阅

droplang, CREATE EXTENSION, CREATE LANGUAGE

createuser

Name

createuser -- 创建一个新的PostgreSQL用户帐户

Synopsis

`createuser` [`_connection-option_` ...] [`_option_` ...] [`_username_`]

描述

createuser创建一个新的PostgreSQL 用户（更准确地说是一个角色）。 只有超级用户和具有 `CREATEROLE` 权限的用户可以创建新的用户。 因此createuser必须由超级用户或者是具有 `CREATEROLE` 权限的用户执行。

如果你想创建一个新的超级用户，你必须以超级用户身份连接， 而不仅仅是一个有 `CREATEROLE` 权限的用户。 成为超级用户就意味着将在数据库里绕开所有访问权限检查，因此，不要轻易授予超级用户权限。

createuser是对 SQL命令CREATE ROLE的封装。 因此，用哪种方法创建的用户都一样。

选项

createuser接受下列命令行参数：

`_username_`

指定要创建的PostgreSQL用户名称。 此名称必须与本 PostgreSQL安装的所有现有角色不同。

`-c` `_number_``--connection-limit=``_number_`

为新用户设置最大数目的连接限制。缺省为无限制。

`-d` `--createdb`

允许该新用户创建数据库。

`-D` `--no-createdb`

禁止该新用户创建数据库。这是缺省（设置）。

`-e` `--echo`

回显createuser生成并发送到服务端的命令。

`-E` `--encrypted`

加密存储在数据库中的用户的密码。 如果没有指定，使用缺省设置。

`-i` `--inherit`

该新角色将自动继承其所属角色组的权限。这是缺省（设置）。

`-I` `--no-inherit`

该新角色将不会自动继承他所属角色组的权限。

`--interactive`

若没有在命令行上指定用户名，提示缺少用户名。并且 `-d` / `-D`，`-r` / `-R`，`-s` / `-S` 选项任何属性没有在命令行指定，也会提示缺少属性（而不会使用缺省值）。 （PostgreSQL 9.1以前这是缺省的设置。）

`-l` `--login`

该新用户将允许登录（也就是说，用户名可以作为初始会话的用户标识符）。这是缺省（设置）。

`-L` `--no-login`

该新用户将不允许登录。 （作为管理数据库权限的手段，（设置）没有登录权限的角色仍然是有必要的。）

`-N` `--unencrypted`

不加密存储在数据库中的用户的密码。 如果没有指定，使用缺省设置

`-P` `--pwprompt`

如果给出（此选项），createuser将提示输入新用户的密码。 如果不准备使用密码认证方式，那么没必要这么做。

`-r` `--createrole`

该新用户将允许创建新角色 （即，该用户将拥有 `CREATEROLE` 权限）。

`-R` `--no-createrole`

该新用户将不能创建新角色。这是缺省（设置）。

`-s` `--superuser`

该新用户将是一个超级用户。

`-S` `--no-superuser`

该新用户将不是一个超级用户。这是缺省（设置）。

`-V` `--version`

输出createuser命令的版本信息，然后退出。

`--replication`

该新用户将拥有 `REPLICATION` 权限（即角色能启动复制）， 完整的描述见文档CREATE ROLE。

`--no-replication`

该新用户将没有 `REPLICATION` 权限（即角色不能启动复制）， 完整的描述见文档CREATE ROLE。

`-?` `--help`

显示createuser命令的帮助信息，然后退出。

createuser还接受以下命令行选项用于连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务端的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务端侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

进行联接的用户名（不是要创建的用户名）。

`-w` `--no-password`

永远不提示输入密码。 如果服务器要求密码验证和密码 （并且）通过其他方式如 `.pgpass` 文件（验证）不可用， 则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制createuser提示输入密码 （用于联接到服务端，而不是新用户的密码）。

这个选项不是必须的，如果服务器要求认证密码 createuser会自动提示需输入密码。 然而，createuser会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

环境变量

`PGHOST` `PGPORT` `PGUSER`

缺省的联接参数

此实用工具，像大多其他的PostgreSQL实用工具， 还使用libpq支持的环境变量(见 Section 31.14)。

诊断

如果有问题，将会显示后端错误信息。参阅CREATE ROLE 和psql获取可能的问题和错误消息的描述。 数据库服务端必须运行在目标主机。此外，前端库 libpq中的所有缺省连接设置和环境变量都将适用。

示例

在缺省数据库服务器上创建一个 `joe` 用户：

```
<samp class="literal">$</samp> <kbd class="literal">createuser joe</kbd>
```

To create a user `joe` on the default database server with prompting for some additional attributes:

```
<samp class="literal">$</samp> <kbd class="literal">createuser --interactive joe</kbd>
<samp class="literal">Shall the new role be a superuser? (y/n)</samp> <kbd class="literal
<samp class="literal">Shall the new role be allowed to create databases? (y/n)</samp> <kb
<samp class="literal">Shall the new role be allowed to create more new roles? (y/n)</samp
```

在主机 `eden` 端口5000的服务端里创建（跟上例）一样 `joe` 用户， 明确指定属性，请看下列命令：

```
<samp class="literal">$</samp> <kbd class="literal">createuser -h eden -p 5000 -S -D -R -
<samp class="literal">CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;<
```

创建超级用户 `joe`， 并立即指定一个密码：

```
<samp class="literal">$</samp> <kbd class="literal">createuser -P -s -e joe</kbd>
<samp class="literal">Enter password for new role:</samp> <kbd class="literal">xyzzy</kbd
<samp class="literal">Enter it again:</samp> <kbd class="literal">xyzzy</kbd>
<samp class="literal">CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPE
```

在上面例子中，虽然新密码在输入的时候实际上并不会回显出来，但是我们还是明确的把它显示出来了。 正如你看到的，密码是加密后发送到客户端。 如果使用选项`--unencrypted`密码将会回显出来 （也可能在服务器日志和其他地方）， 所以在其他人能看到你的屏幕的时候不要使用`-e`选项。

另请参阅

dropuser, CREATE ROLE

dropdb

Name

dropdb -- 删除一个 PostgreSQL 数据库

Synopsis

`dropdb` [`_connection-option_` ...] [`_option_` ...] `_dbname_`

描述

dropdb 删除一个现有 PostgreSQL 数据库。 执行此命令的用户必须是数据库超级用户或者是数据库的所有者。

dropdb是对 SQL命令DROP DATABASE的封装。 因此，用哪种方法删除数据库都一样。

选项

dropdb 接受下列命令行参数：

`_dbname_`

指定要删除的数据库名称

`-e``--echo`

回显dropdb生成并发送到服务端的命令。

`-i` `--interactive`

在做任何破坏性动作前发出确认提示。

`-V` `--version`

输出dropdb命令的版本信息，然后退出。

`--if-exists`

不要抛出如果数据库不存在时发出的错误。

`-?` `--help`

显示dropdb命令的帮助信息，然后退出。

dropdb 还接受以下命令行选项用于连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

进行联接的用户名。

`-w` `--no-password`

永远不提示输入密码。 如果服务器要求密码验证和密码 通过其他方式如 `.pgpass` 文件（验证）不可用，则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制 dropdb 联接到数据库之前提示输入密码。

这个选项不是必须的，如果服务器要求认证密码 dropdb 会自动提示需输入密码。 然而 dropdb 会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

`--maintenance-db=``_dbname_`

指定要删除数据库时联接的数据库名称。 如果没有指定，将使用 `postgres` 数据库； 如果不存在（或是正在被删除的数据库）， 将使用 `template1` 数据库。

环境变量

`PGHOST` `PGPORT` `PGUSER`

缺省的联接参数

此实用工具，像大多其他的PostgreSQL 实用工具， 还使用libpq支持的环境变量 (见 Section 31.14)。

诊断

如果有问题，参阅 DROP DATABASE 和psql 获取可能的问题和错误消息的描述。 数据库服务器必须在目标机器上运行。 此外，前端库 libpq中的所有缺省连接设置和环境变量都将适用。

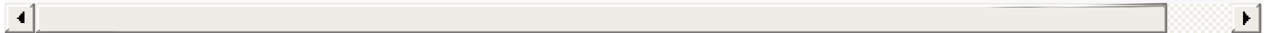
示例

删除缺省数据库服务器上的 `demo` 数据库：

```
<samp class="literal">$</samp> <kbd class="literal">dropdb demo</kbd>
```

在主机 `eden` 上删除 `demo` 数据库，端口是5000， 执行命令前提示并且回显执行的命令：

```
<samp class="literal">$</samp> <kbd class="literal">dropdb -p 5000 -h eden -i -e demo</kb
<samp class="literal">Database "demo" will be permanently deleted.
Are you sure? (y/n)</samp> <kbd class="literal">y</kbd>
<samp class="literal">DROP DATABASE demo;</samp>
```



另请参阅

createdb, DROP DATABASE

droplang

Name

droplang -- 删除一个PostgreSQL过程语言

Synopsis

`droplang` [`_connection-option_` ...] `_langname_` [`_dbname_`]

`droplang` [`_connection-option_` ...] `--list` | `-l` [`_dbname_`]

描述

droplang是一个从 PostgreSQL数据库中删除现有过程语言的工具。

droplang仅是对SQL命令 DROP EXTENSION的封装。

Caution
droplang是一个废弃了的可能在未来 PostgreSQL版本中删除的命令。 建议直接使用 `DROP EXTENSION` 命令。

选项

droplang接受下列命令行参数：

`_langname_`

指定要删除的程序语言的名称。（名称需小写。）

`[-d]` `_dbname_``[--dbname=]``_dbname_`

指定从哪个数据库删除该语言。缺省使用和当前系统用户同名的数据库。

`-e` `--echo`

回显所执行的SQL命令。

`-l` `--list`

显示在目标数据库中已经安装的语言的列表。

`-V` `--version`

输出droplang命令的版本信息，然后退出。

`-?` `--help`

显示droplang命令的帮助信息，然后退出。

droplang还接受以下命令行选项用于联接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

进行联接的用户名。

`-w` `--no-password`

永远不提示输入密码。 如果服务器要求密码验证和密码 （并且）通过其他方式如 `.pgpass` 文件（验证）不可用，则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制droplang联接到数据库之前提示输入密码。

这个选项不是必须的，如果服务器要求认证密码 droplang会自动提示需输入密码。 然而droplang会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

环境变量

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

缺省的联接参数

此实用工具，像大多其他的PostgreSQL实用工具， 还使用 libpq支持的环境变量（见 Section 31.14）。

诊断

大多数错误信息都是自解释的。如果不是，使用`--echo`选项运行droplang然后查看相应的SQL命令的详细信息。 此外，前端库 libpq中的所有缺省联接设置和环境变量都将适用。

注意

使用createlang添加一种语言。

示例

删除`pltcl`语言：

```
<samp class="literal">$</samp> <kbd class="literal">droplang pltcl dbname</kbd>
```

另请参阅

createlang, DROP EXTENSION, DROP LANGUAGE

dropuser

Name

dropuser -- 删除一个PostgreSQL用户账户

Synopsis

`dropuser` [`_connection-option_` ...] [`_option_` ...] [`_username_`]

描述

dropuser删除现有 PostgreSQL用户。 只有超级用户和拥有 `CREATEROLE` 权限的用户可以删除PostgreSQL用户。 （要删除一个超级用户，你必须首先是超级用户。）

dropuser是对 SQL命令DROP ROLE的封装。 因此，用哪种方法删除用户都一样。

选项

dropuser接受下列命令行参数：

`_username_`

指定要删除的PostgreSQL用户名称。 如果没有在命令行上指定用户名称和指定了 `-i` / `--interactive` 选项，将会提示输入一个名称。

`-e``--echo`

回显dropuser生成并发送到服务端的命令。

`-i` `--interactive`

真正删除用户之前提示确认，并且如果没有在命令行指定名称将提示输入用户名。

`-V` `--version`

输出dropuser命令的版本信息，然后退出。

`--if-exists`

不要抛出如果用户不存在时发出的错误。

`-?` `--help`

显示dropuser命令的帮助信息，然后退出。

dropuser还接受以下命令行选项用于连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头则被用作到Unix域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器侦听的TCP端口或一个本地Unix域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

进行联接的用户名（不是要删除的用户名）。

`-w` `--no-password`

永远不提示输入密码。如果服务器要求密码验证和密码 （且）通过其他方式如 `.pgpass` 文件（验证）不可用，则联接尝试将失败。 此选项在不需要用户输入密码的批处理作业和脚本中非常有用。

`-W` `--password`

强制dropuser联接到数据库之前提示输入密码。

这个选项不是必须的，如果服务器要求认证密码 dropuser会自动提示需输入密码。 然而dropuser会浪费一个联接尝试判断出该服务器需要密码。 在某些情况下，这是值得键入 `-W` 以避免多余的联接尝试。

环境变量

`PGHOST` `PGPORT` `PGUSER`

缺省的联接参数

此实用工具，像大多其他的PostgreSQL实用工具， 还使用libpq支持的环境变量(见 Section 31.14)。

诊断

如果有问题，参阅DROP ROLE 和psql获取可能的问题和错误消息的描述。 数据库服务器必须在目标机器上运行。 此外，前端库libpq中的所有缺省连接设置和环境变量都将适用。

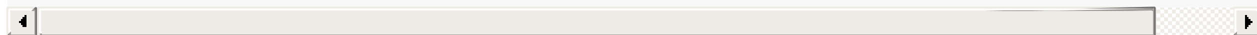
示例

删除缺省数据库服务器上的 `joe` 用户：

```
<samp class="literal">$</samp> <kbd class="literal">dropuser joe</kbd>
```

在主机 `eden` 上删除 `joe` 用户，端口5000， 执行命令前提示并且回显执行的命令：

```
<samp class="literal">$</samp> <kbd class="literal">dropuser -p 5000 -h eden -i -e joe</k
<samp class="literal">Role "joe" will be permanently removed.
Are you sure? (y/n)</samp> <kbd class="literal">y</kbd>
<samp class="literal">DROP ROLE joe;</samp>
```



另请参阅

createuser, DROP ROLE

ecpg

Name

ecpg -- 嵌入的 SQL C 预处理器

Synopsis

`ecpg` [`_option_` ...] `_file_` ...

描述

`ecpg` 是一个嵌入的用于 C 语言的 SQL 预编译器。 它把嵌有 SQL 语句的 C 程序通过将 SQL 调用替换成特殊的函数调用的方法转换成普通的 C 代码。 然后输出的文件就可以用任何 C 编译工具进行处理。

`ecpg` 将把命令行上给出的每个输入文件转换成对应的 C 输出文件。 输入文件最好有 `.pgc` 扩展名，这样，这个扩展将被替换成 `.c` 来决定输出文件名。如果输入文件的扩展不是 `.pgc` ， 那么输出文件名将通过在全文件名后面附加 `.c` 来生成。 输出文件名也可以用 `-o` 选项覆盖。

本手册页并不描述嵌入的 SQL 语句，参阅Chapter 33获更多信息。

选项

`ecpg` 接受下面命令行参数：

`-c`

为 SQL 代码自动生成某种 C 代码。目前，这个选项可以用于 `EXEC SQL TYPE` 。

`-C` `_mode_`

设置一个兼容模式。 `_mode_` 可以是 `INFORMIX` 或 `INFORMIX_SE` 之一。

`-D` `_symbol_`

定义一个 C 预处理器符号。

`-i`

同时也分析系统包含文件。

`-I` `_directory_`

声明一个附加的包含路径。用于寻找通过 `EXEC SQL INCLUDE` 包含的文件。 缺省是：`.`(当前目录)、 `/usr/local/include` 、 在编译时定义的PostgreSQL包含路径(缺省为：`/usr/local/pgsql/include`)、 `/usr/include` 。顺序同上。

`-o` `_filename_`

指定 `ecpg` 应该把它的所有输出写到给出的 `_filename_` 里。

`-r` `_option_`

选择一个运行时行为。 `_Option_` 可以是下列的：

`no_indicator`

不使用指示器，但是使用特殊的值来表示空值。历史上有过数据库使用这种方法。

`prepare`

使用它们之前预备所有语句。Libecpg将保存一个预备语句的缓冲，并且如果再次执行就重复使用一个语句。 如果缓存运行满了，Libecpg将释放最近使用的语句。

`questionmarks`

允许问号标识为了兼容原因作为占位符。很久以前这是缺省。

`-t`

打开自动提交模式。在这种模式下，每个查询都自动提交，除非它是包围在一个明确的事务块中。 在缺省模式下，命令只是在发出 `EXEC SQL COMMIT` 的时候提交。

`-v`

打印额外的信息，包括版本和"include"路径。

`--version`

打印ecpg版本然后退出。

`-?``--help`

显示关于ecpg命令行参数的帮助，然后退出。

注意

在编译预处理的 C 代码文件的时候，编译器需要能够找到PostgreSQL 包含目录里面的ECPG头文件。因此，在调用编译器的时候可能需要使用 `-I` 选项(比如 `-I/usr/local/pgsql/include`)。

使用了嵌入 SQL 的 C 代码必须和 `libecpg` 库链接，比如， 使用这样的链接选项： `-L/usr/local/pgsql/lib -lecpg` 。

这些目录的实际值可以通过pg_config找到。

例子

如果你有一个叫 `prog1.pgc` 的嵌入 SQL 的 C 源代码， 你可以用下面的命令序列创建一个可执行程序：

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

Name

pg_basebackup -- 做一个PostgreSQL 集群的基础备份

Synopsis

`pg_basebackup` [`_选项_` ...]

描述

pg_basebackup用来给一个运行的PostgreSQL 数据库集群进行基础备份。进行时不会影响到连接到数据库的客户端，并且同时可以用于时间点恢复 （参阅Section 24.3）和日志传输或流复制备用服务器的起始点 （参阅Section 25.2）。

pg_basebackup做一个数据库集群文件的二进制拷贝， 同时确保系统自动进出自动备份模式。备份总是使用整个的数据库集群， 不可能只备份单个的数据库或数据库对象。对于单个数据库备份，必须使用如 pg_dump的工具。

备份时通过一个普通的PostgreSQL连接制作的，并且使用复制协议。 该连接必须由超级用户或一个拥有 `REPLICATION` 权限的用户完成 （参阅Section 20.2），并且 `pg_hba.conf` 必须明确允许复制连接。该服务器也必须由max_wal_senders配置， 设置足够高的级别，对于备份至少有一个会话可用。

在同一时刻可能有多个 `pg_basebackup` 运行，但是从性能来说最好只采取一个备份，然后复制结果。

pg_basebackup不止可以从主机备份还可以从备机备份。要从备机备份， 设置备机以使其可以接受复制连接（也就是，设置 `max_wal_senders` 和 hot_standby，并且配置host-based authentication）。 也需要在主机上启用full_page_writes。

请注意，这里有几个从备机在线备份的限制：

- 备份历史文件不是在数据库集群备份时创建的。
- 不保证所有需要备份的WAL文件在备份的最后归档。如果你计划使用备份作为归档恢复， 并希望保证此刻所有需要的文件都可以使用，你需要通过使用 `-x` 选项将他们包含到备份中。
- 如果备机在在线备份期间被提升为主机，则备份失败。

- 所有需要备份的WAL记录必须包含足够的全版书写，这需要你在主机上启用 `full_page_writes` 并且不使用类似pg_compresslog这样的工具作为 `archive_command` 从WAL文件中删除全版书写。

选项

下列的命令行选项控制输出的位置和格式。

`-D` `_directory_``--pgdata=``_directory_`

写输出的目录。pg_basebackup将创建目录和任何父目录（如果需要）。 该目录可能已经存在，但是如果该目录已经存在并且不为空则是一个错误。

当备份在tar模式，并且目录以 `-` （破折号）声明，那么tar文件将写入 `stdout` 。

这个选项是必需的。

`-F` `_format_` `--format=``_format_`

选择输出格式。 `_format_` 可以是下列之一：

`p` `plain`

将输出写作普通的文件，该文件与当前数据目录和表空间有相同的布局。当集群没有额外的表空间时， 整个数据库将被放入目标目录中。如果集群包含额外的表空间，那么主数据目录将被放入目标目录中， 但是所有其他表空间将被放入与它们在服务器上相同的绝对路径中。

这是缺省的格式。

`t` `tar`

将输出写作目标目录中的tar文件。主数据目录将被写成一个命名为 `base.tar` 的文件，所有其他表空间将以表空间OID命名。

如果值 `-` （破折号）被声明为目标目录，那么tar内容将被写成标准输出， 适合于管道如gzip。只有集群没有额外的表空间时这才是可能的。

`-R` `--write-recovery-conf`

在输出目录（或使用tar格式时为基础归档文件）中写一个最小的 `recovery.conf` 以减轻设置一个备用服务器。

`-x` `--xlog`

使用这个选项等同于使用方法 `fetch` 的 `-X` 。

`-X` `_method_` `--xlog-method=``_method_`

在备份中包含所需的事务日志文件（WAL文件）。这将包括所有在备份期间产生的事务日志。 如果声明了这个选项，那么直接在提取出的目录（不需要参考日志归档）中启动一个主进程是可能的。 因此使其成为一个完全独立的备份。

支持下列收集事务日志的方法：

`f` `fetch`

事务日志文件在备份结束时收集。因此，wal_keep_segments 参数有必要设置的足够高，使日志在备份结束之前不会被删除。如果日志在要被转移的时候已经转动了， 那么备份将失败并且不能使用。

`s` `stream`

当备份创建时流事务日志。这将在运行备份时打开又一个到服务器的连接并并行启动流事务日志。 因此，它将使用两个max_wal_senders参数配置的插槽。 只要客户端可以跟上接收到的事务日志，使用这个方法需要没有额外的事务日志在主机上保存。

`-z` `--gzip`

启用tar文件输出的gzip压缩，使用缺省的压缩级别。只有使用tar模式的时候可用压缩。

`-Z` `_level_` `--compress=``_level_`

启用tar文件输出的gzip压缩，并声明压缩级别（1到9，9为顶级压缩）。只有使用tar模式的时候可用压缩。

下列的命令行选项控制备份的生成和程序的运行。

`-c` `_fast|spread_` `--checkpoint=``_fast|spread_`

设置检查点模式为fast或spread（缺省）。

`-l` `_label_` `--label=``_label_`

为备份设置标签。如果没有声明，将使用"`pg_basebackup 基础备份`"的默认值。

`-P` `--progress`

启用进展报告。启用这个选项将会在备份期间发送一个大概的进展报告。因为数据库可能会在备份期间改变， 所以这只是一个大概并且可能不正好是 `100%` 结束。特别是， 当在备份中包含WAL日志时，数据的总量不能提前估计，并且这种情况下一旦它传递的总估计不包含WAL， 那么估计的目标大小还会增加。

当启用这个选项时，备份将从枚举整个数据库的大小开始，然后返回并发送实际的内容。 这样可能会导致备份时间稍长一些，尤其是在发送第一个数据之前的时间稍长。

`-v` `--verbose`

启用冗长模式。将在启动和关闭时输出一些额外的步骤，如果也启用了进度报告， 则也显示当前正在被处理的准确的文件名。

下列的命令行选项控制数据库连接参数。

`-d` `_connstr_` `--dbname=``_connstr_`

声明用来连接到服务器的参数，作为一个连接字符串。参阅Section 31.1.1获取更多信息。

该选项被称为 `--dbname` 是为了与其他客户端应用的一致性，但是因为 pg_basebackup不连接到集群中的任何特别的数据库， 所以将忽略连接字符串中的数据库名字。

`-h` `_host_` `--host=``_host_`

声明服务器正在运行的机器的主机名。如果这个值以一个斜线开始，则被用作Unix域套接字的目录。 默认从 `PGHOST` 环境变量中获取（如果设置了），否则尝试一个Unix域套接字连接。

`-p` `_port_` `--port=``_port_`

声明服务器正在监听的TCP端口或本地Unix域套接字文件扩展。缺省是 `PGPORT` 环境变量（如果设置了），否则是内编译的缺省。

`-s` `_interval_` `--status-interval=``_interval_`

声明状态数据包发送回服务器的秒数。这允许对服务器进程的更简单的监视。为了避免连接超时， 零值完全禁用定期状态更新，尽管服务器需要时仍然发送一个更新。缺省值是10秒。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制pg_basebackup在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则pg_basebackup 自动提示一个密码。不过，pg_basebackup 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

其他选项也可用：

`-V` `--version`

打印pg_basebackup版本然后退出。

`-?` `--help`

显示关于pg_basebackup命令行参数的帮助然后退出。

环境变量

这个工具，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。

注意

备份将包含数据目录和表空间中的所有文件，包括配置文件和任何第三方放置在目录中的额外文件。 数据目录中只允许普通的文件和目录，不允许符号链接或特殊设备文件。

当恢复一个备份时，PostgreSQL管理表空间的方式， 所有额外表空间的路径必须完全相同。 不过， 主数据目录可以重新定位任意位置。

pg_basebackup可以作用于相同版本或更老版本的服务器（最低9.1）， 不过，WAL流模式（-X流）只能是服务器版本9.3。

例子

在 `mydbserver` 创建一个服务器的基础备份，并存储到本地目录 `/usr/local/pgsql/data` 中：

```
<samp class="literal">$</samp> <kbd class="literal">pg_basebackup -h mydbserver -D /usr/l
```

创建一个本地服务器的备份，用一个为每个表空间压缩的tar文件，并存储到目录 `backup` 里，运行时显示进度报告：

```
<samp class="literal">$</samp> <kbd class="literal">pg_basebackup -D backup -Ft -z -P</kb
```

创建一个单表空间本地数据库的备份并使用bzip2压缩：

```
<samp class="literal">$</samp> <kbd class="literal">pg_basebackup -D - -Ft | bzip2 > back
```

如果数据库中有多个表空间那么这个命令将会失败。

又见

pg_dump

pg_config

Name

pg_config -- retrieve information about the installed version of PostgreSQL

Synopsis

`pg_config` [`_option_` ...]

Description

The pg_config utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

Options

To use pg_config, supply one or more of the following options:

`--bindir`

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

`--docdir`

Print the location of documentation files.

`--htmldir`

Print the location of HTML documentation files.

`--includedir`

Print the location of C header files of the client interfaces.

`--pkgincludedir`

Print the location of other C header files.

`--includedir-server`

Print the location of C header files for server programming.

`--libdir`

Print the location of object code libraries.

`--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files might also be installed in this directory.)

`--localedir`

Print the location of locale support files. (This will be an empty string if locale support was not configured when PostgreSQL was built.)

`--mandir`

Print the location of manual pages.

`--sharedir`

Print the location of architecture-independent support files.

`--sysconfdir`

Print the location of system-wide configuration files.

`--pgxs`

Print the location of extension makefiles.

`--configure`

Print the options that were given to the `configure` script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.) See also the examples below.

`--cc`

Print the value of the `CC` variable that was used for building PostgreSQL. This shows the C compiler used.

`--cppflags`

Print the value of the `CPPFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches needed at preprocessing time (typically, `-I` switches).

`--cflags`

Print the value of the `CFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches.

`--cflags_sl`

Print the value of the `CFLAGS_SL` variable that was used for building PostgreSQL. This shows extra C compiler switches used for building shared libraries.

`--ldflags`

Print the value of the `LDFLAGS` variable that was used for building PostgreSQL. This shows linker switches.

`--ldflags_ex`

Print the value of the `LDFLAGS_EX` variable that was used for building PostgreSQL. This shows linker switches used for building executables only.

`--ldflags_sl`

Print the value of the `LDFLAGS_SL` variable that was used for building PostgreSQL. This shows linker switches used for building shared libraries only.

`--libs`

Print the value of the `LIBS` variable that was used for building PostgreSQL. This normally contains `-l` switches for external libraries linked into PostgreSQL.

`--version`

Print the version of PostgreSQL.

`-?``--help`

Show help about pg_config command line arguments, and exit.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Notes

The option `--includedir-server` was added in PostgreSQL 7.2. In prior releases, the server include files were installed in the same location as the client headers, which could be queried with the option `--includedir`. To make your package handle both cases, try the newer option first and test the exit status to see whether it succeeded.

The options `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl`, and `--libs` were added in PostgreSQL 8.1. The option `--htmldir` was added in PostgreSQL 8.4. The option `--ldflags_ex` was added in PostgreSQL 9.0.

In releases prior to PostgreSQL 7.1, before `pg_config` came to be, a method for finding the equivalent configuration information did not exist.

Example

To reproduce the build configuration of the current PostgreSQL installation, run the following command:

```
eval ./configure `pg_config --configure`
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

pg_dump

Name

pg_dump -- 将一个PostgreSQL数据库转储到一个脚本文件或者其它归档文件中

Synopsis

`pg_dump` [`_connection-option_` ...] [`_option_` ...] [`_dbname_`]

描述

pg_dump是一个用于备份PostgreSQL 数据库的工具。它甚至可以在数据库正在使用的时候进行完整一致的备份。 pg_dump并不阻塞其它用户对数据库的访问(读或者写)。

转储格式可以是一个脚本或者归档文件。脚本转储的格式是纯文本，它包含许多 SQL 命令，这些 SQL 命令可以用于重建该数据库并将之恢复到保存成脚本的时候的状态。 使用psql从这样的脚本中恢复。 它们甚至可以用于在其它机器甚至是其它硬件体系的机器上重建该数据库， 通过对脚本进行一些修改，甚至可以在其它 SQL 数据库产品上重建该数据库。

归档文件格式必须和pg_restore一起使用重建数据库。 它们允许pg_restore对恢复什么东西进行选择， 或者甚至是在恢复之前对需要恢复的条目进行重新排序。归档文件也是设计成可以跨平台移植的。

如果一种候选文件格式和pg_restore结合， 那么pg_dump就能提供一种灵活的归档和传输机制。 pg_dump可以用于备份整个数据库，然后就可以使用 pg_restore检查这个归档和/或选择要恢复的数据库部分。 最灵活的输出文件格式是"custom"(自定义)格式(`-Fc`)和"directory"（目录）格式(`-Fd`)。 它们允许对归档元素进行选取和重新排列，支持并行恢复并且缺省时是压缩的。 "directory"格式是唯一支持并行转储的格式。

在运行pg_dump的时候，应该检查输出， 看看是否有任何警告存在(在标准错误上打印)，特别是下面列出的限制。

选项

下面的命令行参数控制输出的内容和格式。

`_dbname_`

将要转储的数据库名。如果没有声明这个参数，那么使用环境变量 `PGDATABASE` 。 如果那个环境变量也没声明，那么使用发起连接的用户名。

`-a``--data-only`

只输出数据，不输出模式(数据定义)。转储表数据、大对象和序列值。

这个选项类似于声明 `--section=data` ，但是只是因为历史原因存在并不完全相同。

`-b` `--blobs`

在转储中包含大对象。除非指定了 `--schema` ， `--table` ， `--schema-only` 开关，否则这是默认行为。因此 `-b` 开关仅用于在选择性转储的时候添加大对象。

`-c` `--clean`

输出命令在输出创建数据库命令之前先清理(drop)该数据库对象。 （如果任何对象在目标数据库中不存在，则转储可能生成一些无害的错误信息。）

这个选项只是对纯文本格式有意义。对于归档格式，可以在调用 `pg_restore` 的时候声明该选项。

`-C` `--create`

以一条创建该数据库本身并且与这个数据库连接命令开头进行输出。如果是这种形式的脚本， 那么你在运行脚本之前和目的安装中的哪个数据库连接就不重要了。如果也声明了 `--clean` ，那么脚本在重新连接到数据库之前删除并重新创建目标数据库。

这个选项只对纯文本格式有意义。对于归档格式，可以在调用 `pg_restore` 的时候声明该选项。

`-E` `_encoding_` `--encoding=``_encoding_`

以指定的字符集编码创建转储。缺省时，转储是按照数据库编码创建的。 另外一个获取同样结果的方法是将 `PGCLIENTENCODING` 环境变量设置为期望的转储编码。

`-f` `_file_` `--file=``_file_`

把输出发往指定的文件。文件基础输出格式时可以省略这个参数，这种情况下使用标准输出。 但是，在声明目标目录而不是文件时必须给出目录输出格式。在这种情况下， 目录通过 `pg_dump` 创建并且必须之前不存在。

`-F` `_format_` `--format=``_format_`

选择输出的格式。 `_format_` 可以是下列之一：

`p` `plain`

纯文本SQL脚本文件(缺省)。

`c` `custom`

适合输入到pg_restore里的自定义格式归档。 加上目录输出格式，这是最灵活的格式，它允许在转储期间对已归档的条目进行手动选择和重新排列。 这个格式缺省的时候是压缩的。

`d` `directory`

适合输入到pg_restore里的目录格式归档。这将创建一个目录， 该目录包含一个为每个被转储的表和二进制大对象的文件，加上一个号称目录的文件， 该文件以pg_restore可读的机器可读格式描述转储的对象。 目录格式归档可以用标准Unix工具操作；例如，在非压缩归档中的文件可以用 gzip工具压缩。这个格式缺省的时候是压缩的， 并且也支持并行转储。

`t` `tar`

适合输入到pg_restore里的 `tar` 归档文件。 tar格式兼容目录格式；提取tar格式归档产生一个有效的目录格式归档。不过， tar格式不支持压缩并且限制单独的表为8 GB。还有，表数据条目的相关顺序在转储期间不能更改。

`-i` `--ignore-version`

一个现在已经不用了的废弃选项。

`-j` `_njobs_` `--jobs=``_njobs_`

通过同时转储 `_njobs_` 表并行运行转储。 该选项减少了转储的时间，但是也增加了数据库服务器的负载。可以只将这个选项用于目录输出格式， 因为这是多进程可以同时写它们的数据的唯一的输出格式。

pg*dump*将打开`_njobs*` + 1个到数据库的连接， 所以确保你的max_connections设置足够高以适应所有的连接。

运行并行转储时在数据库对象上请求排他锁会导致转储失败。原因是pg_dump 主进程在工作进程稍后转储的对象上请求共享锁，这样做是为了确保在转储运行时没有人删除它们或移走它们。 如果另一个客户端然后在一个表上请求一个排他锁，该锁将不被授予，但是将会排队等候主进程的共享锁释放。 因此，任意其他访问表的请求也将不被授予，并且会在排他锁请求后排队。 这包含工作进程尝试转储这个表。没有防范措施这将是一个经典的死锁情况。为了检测这个冲突， pg_dump工作进程请求使用 `NOWAIT` 选项请求另外一个共享锁。 如果没有授予工作进程这个共享锁，那么肯定是另外一个人在此期间请求了一个排他锁， 并且没有办法继续进行转储了，所以pg_dump只能退出转储。

对于一个一致的备份，数据库服务器需要支持同步快照，这是PostgreSQL 9.2 引入的一个特性。有了这个特性，数据库客户端可以保证他们看到相同的数据设置，即使它们使用不同的连接。 `pg_dump -j` 使用多个数据库连接；它与主进程连接到数据库一次，然后再次连接到每个worker工作。 没有同步快照特性，不同的worker工作将不会保证在每个连接中看到相同的数据，这将导致一个不一致的备份。

如果你想在一个9.2之前的服务器上运行并行转储，那么你需要保证在主进程到最后一个worker工作连接到数据库之间， 数据库内容不会改变。最简单的方法是在开始备份之前叫停所有数据修改进程（DDL和DML）访问数据库。 你也需要在9.2之前的PostgreSQL服务器上运行 `pg_dump -j` 时，声明 `--no-synchronized-snapshots` 参数。

`-n` `_schema_` `--schema=``_schema_`

只转储匹配 `_schema_` 的模式内容， 包括模式本身以及其中包含的对象。如果没有声明这个选项， 所有目标数据库中的非系统模式都会被转储出来。可以使用多个 `-n` 选项指定多个模式。 同样， `_schema_` 参数将按照psql 的 `\d` 命令的规则(参见*Patterns*) 被解释为匹配模式， 因此可以使用通配符匹配多个模式。在使用通配符的时候，最好用引号进行界定， 以防止shell 将通配符进行扩展。参阅例子。

> **Note:** 如果指定了 `-n`，那么pg_dump 将不会转储那些模式所依赖的其他数据库对象。 因此， 无法保证转储出来的内容一定能够在另一个干净的数据库中恢复成功。
>
> **Note:** 非模式对象(比如大对象)不会在指定 `-n` 的时候被转储出来。 你可以使用 `--blobs` 明确要求转储大对象。

`-N` `_schema_` `--exclude-schema=``_schema_`

不转储任何匹配 `_schema_` 的模式内容。 模式匹配规则与 `-n` 完全相同。可以指定多个 `-N` 以排除多种匹配的模式。

如果同时指定了 `-n` 和 `-N`，那么将只转储匹配 `-n` 但不匹配 `-N` 的模式。如果出现 `-N` 但是不出现 `-n`， 那么匹配 `-N` 的模式将不会被转储。

`-o` `--oids`

作为数据的一部分，为每个表都输出对象标识(OIDs)。 如果你的应用需要OID字段的话(比如在外键约束中用到)，那么使用这个选项。 否则，不应该使用这个选项。

`-O` `--no-owner`

不把对象的所有权设置为对应源数据库。pg_dump默认发出 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句以设置创建的数据库对象的所有权。如果这些脚本将来没有被超级用户 (或者拥有脚本中全部对象的用户)运行的话将会失败。 `-O` 选项就是为了让该脚本可以被任何用户恢复并且将脚本中对象的所有权赋予该选项指定的用户。

这个选项只是对纯文本格式有意义。对于归档格式，在调用 `pg_restore` 的时候可以声明该选项。

`-R` `--no-reconnect`

这个选项已经过时，但是出于向下兼容的考虑，仍然接受这个选项。

`-s` `--schema-only`

只输出对象定义(模式)，不输出数据。

这个选项与 `--data-only` 相反。类似于，但是由于历史原因不等于声明 `--section=pre-data --section=post-data` 。

不要与 `--schema` 选项混淆， `--schema` 使用不同含义的"schema"。

排除表数据只是数据库中表的一个子集，参阅 `--exclude-table-data` 。

`-S` `_username_` `--superuser=``_username_`

指定关闭触发器时需要用到的超级用户名。它只有使用了 `--disable-triggers` 的时候才有影响。一般情况下最好不要输入这个参数，而是用超级用户启动生成的脚本。

`-t` `_table_` `--table=``_table_`

只转储出匹配 `_table_` 的表（或视图、序列、外表）。 可以使用多个 `-t` 选项匹配多个表。同样， `_table_` 参数将按照psql的 `\d` 命令的规则(参见 *Patterns*)被解释为匹配模式， 因此可以使用通配符匹配多个模式。在使用通配符的时候，最好用引号进行界定， 以防止 shell 将通配符进行扩展。参阅例子。

使用了 `-t` 之后， `-n` 和 `-N` 选项就失效了。因为被 `-t` 选中的表将无视 `-n` 和 `-N` 选项而被转储，同时除了表之外的其他对象不会被转储。

> **Note:** 如果指定了 `-t` ，那么pg_dump 将不会转储任何选中的表依赖的其它数据库对象。因此， 无法保证转储出来的表能在一个干净的数据库中成功恢复。
>
> **Note:** `-t` 选项与PostgreSQL 8.2 之前的版本不兼容。 之前的 `-t tab` 将转储所有名为 `tab` 的表， 但是现在只转储在默认搜索路径中可见的表。写成 `-t '*.tab'` 将等价于老版本的行为。 同样，你必须用 `-t sch.tab` 而不是老版本的 `-n sch -t tab` 选择特定模式中的表。

`-T` `_table_` `--exclude-table=``_table_`

不要转储任何匹配 `_table_` 模式的表。 模式匹配规则与 `-t` 完全相同。可以指定多个 `-T` 以排除多种匹配的表。

如果同时指定了 `-t` 和 `-T` ，那么将只转储匹配 `-t` 但不匹配 `-T` 的表。如果出现 `-T` 但是不出现 `-t` ， 那么匹配 `-T` 的表将不会被转储。

`-v` `--verbose`

指定冗余模式。这样将令pg_dump 输出详细的对象评注以及转储文件的启停时间和进度信息到标准错误上。

`-V` `--version`

打印pg_dump版本然后退出。

`-x` `--no-privileges` `--no-acl`

禁止转储访问权限(grant/revoke 命令)。

`-Z` `_0..9_` `--compress=``_0..9_`

指定要使用的压缩级别。0表示不压缩。对于自定义归档格式，指定单个表数据段的压缩， 并且缺省是中等水平的压缩。对于纯文本输出，设置非零压缩级别会压缩整个输出文件， 就像通过gzip反馈回来一样；但是缺省是不压缩的。 tar归档模式当前不支持压缩。

`--binary-upgrade`

此选项用于在线升级工具。不建议也不支持用于其他目的。该选项的行为可能会在将来的版本中改变。

`--column-inserts` `--attribute-inserts`

把数据转储为带有明确字段名的 `INSERT` 命令 (`INSERT INTO` `_table_` (`_column_` , ...) VALUES ...)。这样会导致恢复非常缓慢，它主要用于制作那种可以用于其它非 PostgreSQL 数据库的转储。由于这个选项为每条记录都生成一条命令， 因此如果其中某一行命令出错，那么将仅有该行数据丢失，而不是整个表的数据丢失。

`--disable-dollar-quoting`

这个选项关闭使用美元符界定函数体。强制它们用 SQL 标准的字符串语法的引号包围。

`--disable-triggers`

这个选项只是和创建仅有数据的转储相关。它告诉pg_dump 包含在恢复数据时临时关闭目标表上触发器的命令。如果在表上有参照完整性检查或者其它触发器， 而恢复数据的时候不想重载他们，那么就应该使用这个选项。

目前，为 `--disable-triggers` 发出的命令必须以超级用户来执行。因此， 你应该同时用 `-S` 声明一个超级用户名， 或者最好是用一个超级用户的身份来启动这个生成的脚本。

这个选项只对纯文本格式有意义。对于归档格式，可以在调用 `pg_restore` 的时候声明这个选项。

`--exclude-table-data=``_table_`

不要转储任何匹配 `_table_` 模式的表。 模式匹配规则与 `-t` 完全相同。可以给出多个 `--exclude-table-data` 以排除多个匹配的表。当你需要指定表的定义时该选项是有用的，即使你不需要表里面的数据。

要排除数据库中所有表的数据，参阅 `--schema-only` 。

`--inserts`

将数据输出为的 `INSERT` 命令(而不是 `COPY`)。 这样会导致恢复非常缓慢。这个选项主要用于制作那种可以用于其它非 PostgreSQL数据库的转储。由于这个选项为每条记录都生成一条命令， 因此如果其中某一行命令出错，那么将仅有该行数据丢失，而不是整个表的数据丢失。请注意，如果你重新排列了字段顺序，那么恢复可能会完全失败。 `--column-inserts` 更安全，但是也更慢。

`--lock-wait-timeout=``_timeout_`

在转储开始的时候不要等待请求一个共享表锁。相反，如果无法在指定的 `_timeout_` 内锁住表则失败。 timeout可以用任意 `SET statement_timeout` 接受的格式声明。 （允许的值依赖于你转储的服务器版本，但是自7.3以来，所有的版本都接受毫秒的整数值。 当从7.3以前的版本服务器中转储时，省略该选项。）

`--no-security-labels`

不转储安全标签。

`--no-synchronized-snapshots`

该选项允许在9.2以前的服务器上运行 `pg_dump -j` ，参阅 `-j` 参数的文档获取更多信息。

`--no-tablespaces`

不要输出选择表空间的命令。有了该选项，所有对象在转储期间都将在缺省的表空间中创建。

这个选项只是对纯文本格式有意义。对于归档格式，在调用 `pg_restore` 的时候可以声明该选项。

`--no-unlogged-table-data`

不要转储未记录表的内容。该选项对于表定义（模式）是否转储没有影响； 它只阻止转储表的数据。当从备用服务器转储时，未记录表中的数据总是排除。

`--quote-all-identifiers`

强制给所有标识符加上引号。这在转储一个数据库到一个可能引入了额外关键字的新版本中时可能是有用的。

`--section=``_sectionname_`

只转储命名的章节。该章节名可以是 `pre-data` , `data` , 或 `post-data` 。可以多次声明这个选项以选择多个章节。 缺省是转储所有章节。

数据章节包含实际的表数据、大对象内容和序列值。原始数据项包含索引、触发器、 规则和约束（除了验证检查约束）的定义。之前的数据项包含所有其他数据定义项。

`--serializable-deferrable`

为转储使用一个 `可串行化` 的事务，以保证使用的快照和稍后的数据库状态一致； 做这些是通过等待事务流中的一个点，该点没有异常会出现，所以不会有转储失败或导致其他事务 `serialization_failure` 而回滚的风险。参阅Chapter 13 获取关于事务隔离和并发控制的更多信息。

这个选项对于只打算灾难恢复的转储没有益处。对于原始数据库仍然在更新时， 加载一个数据库的拷贝作为报告或其他只读加载共享的转储是有帮助的。 没有这个选项，转储会反应一个与任何事务最终提交的序列化执行不一致的状态。 例如，如果使用了批处理技术，可能会在转储中显示一部分，而不是批处理中的所有条目。

在pg_dump开始时，如果没有读写事务在活动，则这个选项没有什么影响。 如果有读写事务在活动，那么转储开始时可能会延迟一段不确定的时间。 一旦运行，有没有开关的性能是一样的。

`--use-set-session-authorization`

输出符合 SQL 标准的 `SET SESSION AUTHORIZATION` 命令而不是 `ALTER OWNER` 命令来确定对象所有权。这样令转储更加符合标准，但是如果转储文件中的对象的历史有些问题， 那么可能不能正确恢复。并且，使用 `SET SESSION AUTHORIZATION` 的转储需要数据库超级用户的权限才能转储成功，而 `ALTER OWNER` 需要的权限则低得多。

`-?` `--help`

显示关于pg_dump命令行参数的帮助然后退出。

下面的命令行参数控制数据库的连接参数。

`-d` `_dbname_` `--dbname=``_dbname_`

声明要连接的数据库名称。相当于在命令行中声明 `_dbname_` 作为第一个非选项参数。

如果这个参数包含一个 `=` 符号或以一个有效的URI前缀 (`postgresql://` 或 `postgres://`)开始， 那么将其看做一个 `conninfo` 字符串。参阅Section 31.1 获取更多信息。

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头，则被用作到 Unix 域套接字的路径。 缺省从 `PGHOST` 环境变量中获取(如果设置了的话)，否则，尝试一个 Unix 域套接字连接。

`-p` `_port_` `--port=``_port_`

指定服务器正在侦听的 TCP 端口或本地 Unix 域套接字文件的扩展(描述符)。 缺省使用 `PGPORT` 环境变量(如果设置了的话)，否则，编译时的缺省值。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制pg_dump在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则pg_dump 自动提示一个密码。不过，pg_dump 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

`--role=``_rolename_`

指定创建转储的角色名。这个选项导致连接到数据库之后pg*dump* 发出一个 `SET ROLE` `_*rolename* 命令。 当认证的用户（通过 -U`指定）缺乏pg_dump所需的权限时是很有用的， 可以转变成有所需权限的角色。一些安装有反对作为超级用户直接登录的政策， 使用这个选项允许转储不违反该政策。

环境变量

`PGDATABASE` `PGHOST` `PGOPTIONS` `PGPORT` `PGUSER`

缺省连接参数。

这个功用，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。

诊断

pg_dump在内部使用 `SELECT` 语句。 如果你运行pg_dump时碰到问题，确认你能够使用像psql这样的程序从数据库选取信息。还有，要申请 任何libpq前端库要使用的缺省连接设置和环境变量。

pg_dump的数据库活动通常由统计收集器收集。 如果不需要，你可以通过 `PGOPTIONS` 或 `ALTER USER` 命令设置参数 `track_counts` 为假。

注意

如果你的数据库给 `template1` 数据库增加了任何你自己的东西， 那么请注意把pg_dump的输出恢复到一个真正空的数据库中； 否则你可能会收到因为重复定义所追加的对象而造成的错误信息。 要制作一个没有任何本地附属物的数据库，可以从 `template0` 而不是 `template1` 拷贝，比如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

在进行纯数据转储并且使用了选项 `--disable-triggers` 的时候， pg_dump发出一些查询先关闭用户表上的触发器， 然后插入数据，插入完成后再打开触发器。如果恢复动作在中间停止， 那么系统表可能就会处于一种错误状态。

tar 归档的成员的大小限制于 8 GB 。这是 tar 文件格式的固有限制。 因此这个格式无法用于任何大小超过这个尺寸的表。 tar 归档和任何其它输出格式的总大小是不受限制的，只是可能会有操作系统的限制。

pg_dump生成的转储文件并不包含优化器用于查询规划决策的统计信息。 因此，恢复完之后，建议在每个已恢复的对象上运行 `ANALYZE` ， 以保证最佳的性能；参阅Section 23.1.3和Section 23.1.6 获取更多信息。转储文件也不包含任何 `ALTER DATABASE ... SET` 命令； 这些设置通过pg_dumpall转储，连同数据库用户和其他安装范围的设置。

因为pg_dump常用于向新版本的PostgreSQL 中传递数据，所以pg_dump的输出预计可以加载到比 pg_dump的版本更新的PostgreSQL 服务器版本中。pg_dump还可以从比它自身版本老的PostgreSQL 服务器中转储。（当前，支持后退到版本7.0的服务器。）不过，pg_dump 不能从比它自身主版本新的PostgreSQL服务器中转储；它会拒绝尝试， 而不是冒险制作一个不可用的转储。另外，它不保证pg_dump 的输出可以加载到一个旧的主版本的服务器中，即使该转储是从那个版本的服务器中而来。 加载转储文件到一个旧的服务器可能需要手动编辑转储文件，以删除不被旧版本理解的语法。

例子

将 `mydb` 数据库转储到一个 SQL 脚本文件：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump mydb > db.sql</kbd>
```

将上述脚本导入一个(新建的)数据库 `newdb` ：

```
<samp class="literal">$</samp> <kbd class="literal">psql -d newdb -f db.sql</kbd>
```

将数据库转储为自定义格式的归档文件

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -Fc mydb > db.dump</kbd>
```

将数据库转储为目录格式归档：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -Fd mydb -f dumpdir</kbd>
```

将数据库转储为目录格式归档，并行5个worker工作：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -Fd mydb -j 5 -f dumpdir</kbd
```

将归档文件导入一个(新建的)数据库 `newdb` ：

```
<samp class="literal">$</samp> <kbd class="literal">pg_restore -d newdb db.dump</kbd>
```

转储一个名为 `mytab` 的表：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -t mytab mydb > db.sql</kbd>
```

转储 `detroit` 模式中所有以 `emp` 开头的表， 但是不包括 `employee_log` 表：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -t 'detroit.emp*' -T detroit.
```

转储所有以 `east` 或 `west` 开头并以 `gsm` 结尾的模式， 但是不包括名字中含有 `test` 模式：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -n 'east*gsm' -n 'west*gsm' -
```

同上，不过这一次使用正则表达式的方法：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -n '(east|west)*gsm' -N '*tes
```

转储所有数据库对象，但是不包括名字以 `ts_` 开头的表：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -T 'ts_*' mydb > db.sql</kbd>
```

在 `-t` 等选项中指定大写字母或大小写混合的名字必须用双引号界定， 否则将被自动转换为小写(参见*Patterns*)。 但是因为双引号在 shell 中有特殊含义，所以必须将双引号再放进单引号中。 这样一来，要转储一个大小写混合的表名，你就需要像下面这样：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -t "\"MixedCaseName\"" mydb >
```

又见

pg_dumpall, pg_restore, psql

pg_dumpall

Name

pg_dumpall -- 将一个PostgreSQL数据库集群转储到一个脚本文件中

Synopsis

`pg_dumpall` [`_connection-option_` ...] [`_option_` ...]

描述

pg_dumpall可以转储一个数据库集群里的所有数据库到一个脚本文件。 该脚本文件包含可以用于作为psql输入的SQL命令， 从而恢复数据库。它通过对数据库集群里的每个数据库调用pg_dump 实现这个功能。pg_dumpall还转储出所有数据库公用的全局对象。 而pg_dump并不保存这些对象。 这些信息包括数据库用户和组、表空间，以及性能如适用于整个数据库的访问权限。

因为pg_dumpall从所有数据库中读取表， 所以你很可能需要以数据库超级用户的身份连接，这样才能生成完整的转储。同样， 你也需要超级用户的权限执行保存下来的脚本，这些才能增加用户和组，以及创建数据库。

SQL 脚本将写出到标准输出。使用[-f|file]选项或 shell 操作符把它重定向到文件。

pg_dumpall需要和PostgreSQL 服务器连接多次(每个数据库一次)。如果你使用口令认证，可能每次都会询问口令。 这种情况下写一个 `~/.pgpass` 可能会比较方便。 参阅Section 31.15获取更多信息。

选项

下列命令行参数用于控制输出内容和格式：

`-a``--data-only`

只转储数据，不转储模式(数据定义)。

`-c` `--clean`

在转储结果中包含那些重建之前清理(drop)数据库对象的 SQL 命令。 对规则和表空间的 `DROP` 也会添加进来。

`-f` `_filename_` `--file=``_filename_`

发送输出到指定的文件。如果省略了这个选项，就使用标准输出。

`-g` `--globals-only`

只转储全局对象(角色和表空间)，而不转储数据库。

`-i` `--ignore-version`

一个现在已经忽略了的废弃选项。

`-o` `--oids`

作为数据的一部分，为每个表都输出对象标识(OID)。 如果你的应用需要OID字段的话(比如在外键约束中用到)，那么使用这个选项。 否则，不应该使用这个选项。

`-O` `--no-owner`

不把对象的所有权设置为对应源数据库。pg_dumpall默认发出 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句以设置创建的数据库对象的所有权。如果这些脚本将来没有被超级用户 (或者拥有脚本中全部对象的用户)运行的话将会失败。 `-O` 选项就是为了让该脚本可以被任何用户恢复并且将脚本中对象的所有权赋予该选项指定的用户。

`-r` `--roles-only`

只转储角色，不转储数据库或表空间。

`-s` `--schema-only`

只输出对象定义(模式)，不输出数据。

`-S` `_username_` `--superuser=``_username_`

指定关闭触发器时需要用到的超级用户名。它只有使用了 `--disable-triggers` 的时候才有影响。一般情况下最好不要输入这个参数，而是用超级用户启动生成的脚本。

`-t` `--tablespaces-only`

只转储表空间，不转储数据库或角色。

`-v` `--verbose`

指定冗余模式。这样将令pg_dumpall 输出转储文件的启停时间和进度信息到标准错误上。它将同时启用pg_dump的冗余输出。

`-V` `--version`

打印pg_dumpall的版本然后退出。

`-x` `--no-privileges` `--no-acl`

禁止转储访问权限(grant/revoke 命令)。

`--binary-upgrade`

这个选项用于本地升级工具。不建议也不支持用于其他目的。 该选项的性能可能会在将来的版本中改变。

`--column-inserts` `--attribute-inserts`

把数据转储为带有明确字段名的 `INSERT` 命令(`INSERT INTO` `_table_` (`_column_` , ...) VALUES ...)。这样会导致恢复非常缓慢，它主要用于制作那种可以用于其它非 PostgreSQL 数据库的转储。

`--disable-dollar-quoting`

这个选项关闭使用美元符界定函数体。强制它们用 SQL 标准的字符串语法的引号包围。

`--disable-triggers`

这个选项只是和创建仅有数据的转储相关。它告诉pg_dumpall 包含在恢复数据时临时关闭目标表上触发器的命令。如果在表上有参照完整性检查或者其它触发器， 而恢复数据的时候不想重载他们，那么就应该使用这个选项。

目前，为 `--disable-triggers` 发出的命令必须以超级用户来执行。因此， 你应该同时用 `-S` 声明一个超级用户名， 或者最好是用一个超级用户的身份来启动这个生成的脚本。

`--inserts`

把数据转储为 `INSERT` 命令(而不是 `COPY`)。 这样将令恢复过程非常缓慢，这个选项主要用于制作那种可以用于其它非 PostgreSQL数据库的转储。请注意， 如果你重新排列了字段顺序，那么恢复可能会完全失败。 `--column-inserts` 更安全，但是也更慢。

`--lock-wait-timeout=``_timeout_`

在转储开始的时候不要等待请求一个共享表锁。相反，如果无法在指定的 `_timeout_` 内锁住表则失败。 timeout可以用任意 `SET statement_timeout` 接受的格式声明。 允许的值依赖于你转储的服务器版本，但是自7.3以来，所有的版本都接受毫秒的整数值。 当从7.3以前的版本服务器中转储时，省略该选项。

`--no-security-labels`

不转储安全标签。

`--no-tablespaces`

不要输出为对象创建表空间或选择表空间的命令。有了该选项， 所有对象在转储期间都将在缺省的表空间中创建。

`--no-unlogged-table-data`

不要转储未记录表的内容。该选项对于表定义（模式）是否转储没有影响；它只阻止转储表的数据。

`--quote-all-identifiers`

强制给所有标识符加上引号。这在转储一个数据库到一个可能引入了额外关键字的新版本中时可能是有用的。

`--use-set-session-authorization`

输出符合 SQL 标准的 `SET SESSION AUTHORIZATION` 命令而不是 `ALTER OWNER` 命令来确定对象所有权。这样令转储更加符合标准，但是如果转储文件中的对象的历史有些问题， 那么可能不能正确恢复。

`-?` `--help`

显示关于pg_dumpall命令行参数的帮助然后退出。

下面的命令行参数控制数据库的连接参数。

`-d` `_connstr_` `--dbname=``_connstr_`

指定用于连接到服务器的参数，作为连接字符串。参阅Section 31.1.1获取更多信息。

为了与其他客户端应用保持一致，这个选项被叫做 `--dbname` ， 但是因为 pg_dumpall需要连接到多个数据库，所以连接字符串中的数据库名将会省略。 使用 `-l` 选项指定用于转储全局对象的数据库名和找出应该转储的其他数据库。

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头，则被用作到 Unix 域套接字的路径。 缺省从 `PGHOST` 环境变量中获取(如果设置了的话)，否则，尝试一个 Unix 域套接字连接。

`-l` `_dbname_` `--database=``_dbname_`

指定用于转储全局对象的数据库名和找出应该转储的其他数据库。如果没有声明， 将使用 `postgres` 数据库，如果 `postgres` 数据库不存在， 则使用 `template1` 。

`-p` `_port_` `--port=``_port_`

指定服务器正在侦听的 TCP 端口或本地 Unix 域套接字文件的扩展(描述符)。 缺省使用 `PGPORT` 环境变量(如果设置了的话)，否则，编译时的缺省值。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制pg_dumpall在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则pg_dumpall 自动提示一个密码。不过，pg_dumpall 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-w` 以避免额外的连接尝试。

请注意，密码提示将在每个要转储的数据库上发生。通常，建立一个 `~/.pgpass` 文件比依赖于手动输入密码好的多。

`--role=``_rolename_`

指定创建转储的角色名。这个选项导致连接到数据库之后pg*dumpall* 发出一个 `SET ROLE` `_*rolename* `命令。 当认证的用户（通过` -U`指定）缺乏pg_dumpall所需的权限时是很有用的， 可以转变成有所需权限的角色。一些安装有反对作为超级用户直接登录的政策， 使用这个选项允许转储不违反该政策。

环境变量

`PGHOST` `PGOPTIONS` `PGPORT` `PGUSER`

缺省连接参数。

这个功用，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。

注意

因为pg_dumpall在内部调用pg_dump， 所以，一些诊断信息可以参考pg_dump。

恢复完之后，建议在每个已恢复的对象上运行 `ANALYZE` 。 这样优化器就可以得到有用的统计。你也可以用 `vacuumdb -a -z` 清理所有数据库。

pg_dumpall要求所有需要的表空间目录在进行恢复之前就必须存在， 否则在非标准位置创建数据库将会失败。

例子

转储所有数据库：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dumpall > db.out</kbd>
```

从该文件中恢复数据库：

```
<samp class="literal">$</samp> <kbd class="literal">psql -f db.out postgres</kbd>
```

执行这个命令的时候连接到哪个数据库无关紧要，因为pg_dumpall 创建的脚本将会包含恰当的创建和连接数据库的命令。

又见

看看pg_dump获取可能的错误条件的详细信息。

pg_isready

Name

pg_isready -- check the connection status of a PostgreSQL server

Synopsis

`pg_isready` [`_connection-option_` ...] [`_option_` ...]

Description

pg_isready is a utility for checking the connection status of a PostgreSQL database server. The exit status specifies the result of the connection check.

Options

`-d` `_dbname_``--dbname=``_dbname_`

Specifies the name of the database to connect to.

If this parameter contains an `=` sign or starts with a valid URI prefix (`postgresql://` or `postgres://`), it is treated as a `conninfo` string. See Section 31.1.1 for more information.

`-h` `_hostname_` `--host=``_hostname_`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

`-p` `_port_` `--port=``_port_`

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

`-q` `--quiet`

Do not display status message. This is useful when scripting.

`-t` `_seconds_` `--timeout=``_seconds_`

The maximum number of seconds to wait when attempting connection before returning that the server is not responding. Setting to 0 disables. The default is 3 seconds.

`-U` `_username_` `--username=``_username_`

Connect to the database as the user `_username_` instead of the default.

`-V` `--version`

Print the pg_isready version and exit.

`-?` `--help`

Show help about pg_isready command line arguments, and exit.

Exit Status

pg_isready returns `0` to the shell if the server is accepting connections normally, `1` if the server is rejecting connections (for example during startup), `2` if there was no response to the connection attempt, and `3` if no attempt was made (for example due to invalid parameters).

Environment

`pg_isready`, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Notes

The options `--dbname` and `--username` can be used to avoid gratuitous error messages in the logs, but are not necessary for proper functionality.

Examples

Standard Usage:

```
<samp class="literal">$</samp> <kbd class="literal">pg_isready</kbd>
<samp class="literal">/tmp:5432 - accepting connections</samp>
<samp class="literal">$</samp> <kbd class="literal">echo $?</kbd>
<samp class="literal">0</samp>
```

Running with connection parameters to a PostgreSQL cluster in startup:

```
<samp class="literal">$</samp> <kbd class="literal">pg_isready -h localhost -p 5433</kbd>
<samp class="literal">localhost:5433 - rejecting connections</samp>
<samp class="literal">$</samp> <kbd class="literal">echo $?</kbd>
<samp class="literal">1</samp>
```

Running with connection parameters to a non-responsive PostgreSQL cluster:

```
<samp class="literal">$</samp> <kbd class="literal">pg_isready -h someremotehost</kbd>
<samp class="literal">someremotehost:5432 - no response</samp>
<samp class="literal">$</samp> <kbd class="literal">echo $?</kbd>
<samp class="literal">2</samp>
```

pg_receivexlog

Name

pg_receivexlog -- PostgreSQL集群中的流事务日志

Synopsis

`pg_receivexlog` [`_option_` ...]

描述

pg_receivexlog用于从一个运行的PostgreSQL 集群中流事务日志。该事务日志流使用流复制协议，并写入到文件的本地目录中。 这个目录可以被用作使用时间点恢复的归档目录（参阅Section 24.3）。

pg_receivexlog在服务器上产生事务日志时实时流事务日志， 并且不像archive_command那样等待段的完成。因为这个原因， 当使用pg_receivexlog时不需要设置archive_timeout。

事务日志在普通的PostgreSQL连接上流出，并且使用复制控制。 必须由超级用户或有 `REPLICATION` 权限的用户（参阅 Section 20.2）连接，并且 `pg_hba.conf` 必须明确的允许复制连接。服务器也必须配置的max_wal_senders足够高， 使流至少有一个会话可用。

如果失去连接，或者不能初步建立，带有一个非致命错误，pg_receivexlog 将无限的重试连接，并尽快重建流。要避免这种行为，使用 `-n` 参数。

选项

下列的命令行选项控制输出的位置和格式。

`-D` `_directory_``--directory=``_directory_`

写输出的目录。

这个参数是必需的。

下列的命令行选项控制程序的运行。

`-n` `--no-loop`

不要循环连接错误。相反，立即带有错误退出。

`-v` `--verbose`

启用冗余模式。

下列的命令行选项控制数据库连接参数。

`-d` `_connstr_` `--dbname=``_connstr_`

声明用来连接到服务器的参数，作为一个连接字符串。参阅Section 31.1.1获取更多信息。

该选项被称为 `--dbname` 是为了与其他客户端应用的一致性，但是因为 pg_receivexlog不连接到集群中的任何特别的数据库， 所以将忽略连接字符串中的数据库名字。

`-h` `_host_` `--host=``_host_`

声明服务器正在运行的机器的主机名。如果这个值以一个斜线开始，则被用作Unix域套接字的目录。 默认从 `PGHOST` 环境变量中获取（如果设置了），否则尝试一个Unix域套接字连接。

`-p` `_port_` `--port=``_port_`

声明服务器正在监听的TCP端口或本地Unix域套接字文件扩展。缺省是 `PGPORT` 环境变量（如果设置了），否则是内编译的缺省。

`-s` `_interval_` `--status-interval=``_interval_`

声明状态数据包发送回服务器的秒数。这允许对服务器进程的更简单的监视。为了避免连接超时， 零值完全禁用定期状态更新，尽管服务器需要时仍然发送一个更新。缺省值是10秒。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制pg_receivexlog在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则pg_receivexlog 自动提示一个密码。不过，pg_receivexlog 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

其他选项也可用：

`-V` `--version`

打印pg_receivexlog版本然后退出。

`-?` `--help`

显示关于pg_receivexlog命令行参数的帮助然后退出。

环境变量

这个工具，类似大多数其他PostgreSQL实用工具，也使用由libpq支持的环境变量（参阅Section 31.14）。

注意

当使用pg_receivexlog而不是archive_command时，服务器将持续回收事务日志文件，即使备份没有适当的归档，因为这里没有失败的命令。可以通过在文件还未适当的归档时有一个archive_command失败来绕开：

```
archive_command = 'sleep 5 && test -f /mnt/server/archivedir/%f'
```

最初的时限是必须的，因为pg_receivexlog使用异步复制工作，并且因此稍微落后于主机。

例子

要从服务器 `mydbserver` 流事务日志，并将其存储到本地目录 `/usr/local/pgsql/archive` 中：

```
<samp class="literal">$</samp> <kbd class="literal">pg_receivexlog -h mydbserver -D /usr/
```

又见

pg_basebackup

pg_restore

Name

pg_restore -- 从pg_dump创建的备份文件中恢复PostgreSQL数据库

Synopsis

`pg_restore` [`_connection-option_` ...] [`_option_` ...] [`_filename_`]

描述

pg_restore用于恢复由pg_dump 转储的任何非纯文本格式中的PostgreSQL数据库。 它将发出必要的命令重建数据库，并把它恢复成转储时的样子。 归档(备份)文件还允许pg_restore有选择地进行恢复， 甚至在恢复前重新排列条目的顺序。归档的文件设计成可以在不同的硬件体系之间移植。

pg_restore可以按照两种模式操作。如果声明了数据库名字， 那么pg_restore连接到那个数据库并直接恢复归档内容到数据库里。 否则，先创建一个包含重建数据库所必须的 SQL 命令的脚本，并且写入到一个文件或者标准输出。 这个脚本输出等效于pg_dump的纯文本输出格式。因此， 一些控制输出的选项就是模拟pg_dump的选项设置的。

显然，pg_restore无法恢复那些不存在归档文件中的信息；比如， 如果归档是用"把数据转储为 `INSERT` 命令"选项制作的， 那么pg_restore将不能使用 `COPY` 语句加载数据。

选项

pg_restore接受下列命令行参数：

`_filename_`

要恢复的备份文件（或目录，对于目录格式归档）的位置。如果没有声明，则使用标准输入。

`-a``--data-only`

只恢复数据，而不恢复表模式(数据定义)。恢复表数据、大对象和序列值，如果在档案中存在。

这个选项类似于，但是由于历史原因不等于声明 `--section=data` 。

`-c` `--clean`

创建数据库对象前先清理(删除)它们。（如果任一对象不在目标数据库中， 这可能会产生一些无害的错误信息。）

`-C` `--create`

在恢复数据库之前先创建它。如果也声明了 `--clean`， 那么在连接到数据库之前删除并重建目标数据库。

如果出现了这个选项，和 `-d` 在一起的数据库名只是用于发出最初的 `DROP DATABASE` 和 `CREATE DATABASE` 命令。 所有数据都恢复到名字出现在归档中的数据库中去。

`-d` `_dbname_` `--dbname=``_dbname_`

与数据库 `_dbname_` 连接并且直接恢复到该数据库中。

`-e` `--exit-on-error`

如果在向数据库发送 SQL 命令的时候碰到错误，则退出。 缺省是继续执行并且在恢复结束时显示一个错误计数。

`-f` `_filename_` `--file=``_filename_`

指定生成的脚本的输出文件，或者出现 `-l` 选项时用于列表的文件， 缺省是标准输出。

`-F` `_format_` `--format=``_format_`

指定备份文件的格式。因为pg_restore会自动判断格式， 所以如果一定要指定的话，它可以是下面之一：

`c` `custom`

备份的格式是来自pg_dump的自定义格式。

`d` `directory`

备份是一个目录归档。

`t` `tar`

备份是一个 `tar` 归档。

`-i` `--ignore-version`

一个现在已经忽略了的已废弃的选项。

`-I` `_index_` `--index=``_index_`

只恢复命名的索引。

`-j` `_number-of-jobs_` `--jobs=``_number-of-jobs_`

运行pg_restore耗时最多的部分，该部分使用多重并发工作加载数据、 创建索引或创建约束。这个选项可以显著的减少恢复一个大数据库到一个运行多重处理器服务器的时间。

每个工作是一个过程或一个线程，取决于操作系统，并使用一个单独到服务器的连接。

该选项的最佳值依赖于服务器、客户端和网络的硬件设置。因素包含CPU内核的数量和磁盘设置。 良好的开端是服务器上CPU内核的数量，但是较大的值在许多情况下也可以更快的恢复。 当然，太高的值因为超负荷将会导致性能降低。

该选项只支持自定义和目录归档格式。输入必须是有规律的文件或目录（例如不是通道）。 发出一个脚本而不是连接目录到一个数据库服务器时忽略该选项。还有， 多重工作不能与`--single-transaction`选项一起使用。

`-l` `--list`

列出备份的内容。这个操作的输出可以用作`-L`选项的输入。 请注意，如果过滤开关如`-n`或`-t`和`-l`一起使用， 它们将限制列出的条目。

`-L` `_list-file_` `--use-list=``_list-file_`

以它们在文件中出现的顺序只恢复在`_list-file_`里面的元素。 请注意，如果过滤开关如`-n`或`-t`和`-l`一起使用， 它们将进一步的限制恢复的条目。

`_list-file_`通常通过编辑先前`-l`操作的输出来创建。你可以移动或删除各个行并且也可以通过在行开头放置分号(`;`)的方式注释。例子见下文。

`-n` `_namespace_` `--schema=``_schema_`

只恢复指定名字的模式里面的对象。这个选项可以和`-t`选项一起使用， 实现只转储一个表的数据。

`-O` `--no-owner`

不要输出设置对象权限与最初数据库匹配的命令。缺省时，pg_restore 发出`ALTER OWNER`或`SET SESSION AUTHORIZATION`语句设置创建出来的模式元素的所有者权限。 如果最初的数据库连接不是由超级用户 (或者是拥有所有创建出来的对象的同一个用户)发起的，那么这些语句将失败。 如果使用`-O`，那么任何用户都可以用于初始的连接， 并且这个用户将拥有所有创建出来的对象。

`-P` `_function-name(argtype [, ...])_` `--function=``_function-name(argtype [, ...])_`

只恢复指定的命名函数。请注意仔细拼写函数名及其参数，应该和转储的内容列表中的完全一样。

`-R` `--no-reconnect`

这个选项已经废弃了，但是为了保持向下兼容仍然接受。

`-s` `--schema-only`

只恢复表结构(数据定义)，不恢复数据，在这个意义上来说在归档里有模式的记录。

这个选项是 `--data-only` 的相反。它类似于，但是因为历史原因不等于声明 `--section=pre-data --section=post-data` 。

请不要和 `--schema` 选项混淆，那里使用了"模式"(schema)的其它含义。

`-S` `_username_` `--superuser=``_username_`

设置关闭触发器时使用的超级用户的用户名。只有在设置了 `--disable-triggers` 的时候才有用。

`-t` `_table_` `--table=``_table_`

只恢复指定的表的定义和/或数据。可以声明多个 `-t` 指定多个表。 可以和 `-n` 选项组合以声明一个模式。

`-T` `_trigger_` `--trigger=``_trigger_`

只恢复指定的触发器。

`-v` `--verbose`

声明冗余模式。

`-V` `--version`

打印pg_restore的版本然后退出。

`-x` `--no-privileges` `--no-acl`

禁止恢复访问权限(grant/revoke 命令)。

`-1` `--single-transaction`

将整个恢复过程作为一个完整的事务来执行，也就是将所有恢复命令放在 `BEGIN` / `COMMIT` 之间。这将保证恢复要么全部成功要么没有任何影响。 该选项隐含 `--exit-on-error` 。

`--disable-triggers`

这个选项只有在执行仅恢复数据的时候才相关。它告诉pg_restore 在加载数据的时候执行一些命令临时关闭在目标表上的触发器。 如果你在表上有完整性检查或者其它触发器，而你又不希望在加载数据的时候激活它们， 那么可以使用这个选项。

目前，为 `--disable-triggers` 发出的命令必须以超级用户发出。因此， 你应该也要用 `-S` 声明一个超级用户名，或者更好是以超级用户身份运行 pg_restore。

`--no-data-for-failed-tables`

缺省时，即使创建表的命令因为该表已经存在而失败了，表中的数据仍将被恢复。 使用这个选项之后，这些表的数据就将跳过恢复操作。 如果目标数据库已经包含所需恢复的某些表的内容时，该选项就很有用处了。 比如，用于PostgreSQL扩展的辅助表(例如PostGIS) 就可能已经在目标数据库中恢复过了， 使用该选项就可以防止多次恢复以致重复或者覆盖了已经恢复的数据。

该选项仅在直接向一个数据库中恢复的时候有效，在生成 SQL 脚本输出时无效。

`--no-security-labels`

不要输出恢复安全标签的命令，即使归档包含它们。

`--no-tablespaces`

不要输出选择表空间的命令。有这个选项，在恢复期间所有的对象都将在缺省表空间中创建。

`--section=``_sectionname_`

值恢复指定的章节。章节名可以是 `pre-data` , `data` , 或 `post-data` 。可以多次声明这个选项以选择多个章节。缺省是恢复所有章节。

数据章节包含实际的表数据和大对象定义。原始数据项包含索引、触发器、规则和约束 （除了验证检查约束）的定义。先前的数据项包含所有其他数据定义项。

`--use-set-session-authorization`

输出 SQL 标准的 `SET SESSION AUTHORIZATION` 命令，而不是 `ALTER OWNER` 命令来确定对象的所有权。这样令转储与标准兼容的更好， 但是根据转储中对象的历史，这个转储可能不能恰当地恢复。

`-?` `--help`

显示关于pg_restore命令行参数的帮助然后退出。

pg_restore还接受下面的命令行参数做为连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头，则被用作到 Unix 域套接字的路径。 缺省从 `PGHOST` 环境变量中获取(如果设置了的话)，否则， 尝试一个 Unix 域套接字连接。

`-p` `_port_` `--port=``_port_`

指定服务器正在侦听的 TCP 端口或本地 Unix 域套接字文件的扩展(描述符)。 缺省使用 `PGPORT` 环境变量(如果设置了的话)，否则，编译时的缺省值。

`-U` `_username_` `--username=``_username_`

要连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制pg_restore在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则pg_restore 自动提示一个密码。不过，pg_restore 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

`--role=``_rolename_`

指定执行转储的角色名。这个选项导致连接到数据库之后pg*restore* 发出一个 `SET ROLE` `*rolename* 命令。 当认证的用户（通过 -U`指定）缺乏pg_restore所需的权限时是很有用的， 可以转变成有所需权限的角色。一些安装有反对作为超级用户直接登录的政策， 使用这个选项允许转储不违反该政策。

环境变量

`PGHOST` `PGOPTIONS` `PGPORT` `PGUSER`

缺省连接参数。

这个功用，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。 不过，当没有提供数据库名字时，并不读取 `PGDATABASE` 。

诊断

当使用 `-d` 选项声明了直接数据库连接时， pg_restore在内部执行SQL语句。 如果你运行pg_restore出了毛病， 请确保你能用类似psql这样的东西从数据库中选取信息。 还有，将会应用libpq前端库使用的任何缺省连接设置和环境变量。

注意

如果你的安装给 `template1` 数据库增加了任何你自己的东西， 那么请注意把pg_restore的输出恢复到一个真正空的数据库中； 否则你可能会收到因为重复定义所追加的对象而造成的错误信息。 要制作一个没有任何本地附属物的数据库，可以从 `template0` 而不是 `template1` 拷贝，比如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_restore的局限如下：

- 当向一个已经存在的表恢复数据，并且还使用了 `--disable-triggers` 选项时，pg_restore在插入数据前放出一些查询关闭用户表上的触发器， 在数据插入完成后重新打开它们。如果恢复的中途停止，那么系统表可能处于错误状态。
- pg_restore不能选择性的的恢复大对象。 例如，只恢复指定的表。如果一个归档包含大对象，那么所有大对象都将被恢复， 或如果他们通过 `-L` 、 `-t` 或其他选项排除则一个也不恢复。

参阅pg_dump的文档获取有关pg_dump的局限的细节。

一旦完成恢复，最好在每个恢复的对象上运行 `ANALYZE` ， 以便给优化器有用的统计。参阅Section 23.1.3 和Section 23.1.6获取更多信息。

例子

假定我们已经转储了 `mydb` 数据库到一个自定义格式的文件中：

```
<samp class="literal">$</samp> <kbd class="literal">pg_dump -Fc mydb > db.dump</kbd>
```

删除该数据库并从转储中重建：

```
<samp class="literal">$</samp> <kbd class="literal">dropdb mydb</kbd>
<samp class="literal">$</samp> <kbd class="literal">pg_restore -C -d postgres db.dump</kb
```

在 `-d` 中指定的数据库可以是当前集群中的任意数据库；pg_restore 仅用该名字来为 `mydb` 发出 `CREATE DATABASE` 命令。 使用 `-C` 可以确保数据总是会被恢复到转储文件中指定名字的数据库里面。

将转储出来的数据重新加载到一个新建的数据库 `newdb` 中：

```
<samp class="literal">$</samp> <kbd class="literal">createdb -T template0 newdb</kbd>
<samp class="literal">$</samp> <kbd class="literal">pg_restore -d newdb db.dump</kbd>
```

注意，这里没有使用 `-C` 选项，而是直接链接到将要恢复的数据库上。 还要注意的是，我们从 `template0` 而不是 `template1` 创建了新数据库以确保干净。

要对项目重新排序，首先必须转储归档的目录：

```
<samp class="literal">$</samp> <kbd class="literal">pg_restore -l db.dump > db.list</kbd>
```

这个文件由一行头和每个条目一行组成，比如：

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

这里分号是注释分隔符，而行开头的数字代表赋给每个项目的内部归档 ID 。

文件内的行可以注释、删除和/或重新排列。比如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

可以用做pg_restore的输入并且只会恢复项目 10 和 6 (以这个顺序)：

```
<samp class="literal">$</samp> <kbd class="literal">pg_restore -L db.list db.dump</kbd>
```

又见

pg_dump, pg_dumpall, psql

psql

Name

psql -- PostgreSQL交互终端

Synopsis

`psql` [`_option_` ...] [`_dbname_` [`_username_`]]

描述

psql是一个以终端为基础的 PostgreSQL前端。 它允许你交互地键入查询，然后把它们发送给 PostgreSQL， 再显示查询的结果。 另外，输入可以来自一个文件。还有，它提供了一些元命令和多种类似shell 的特性来实现书写脚本以及对大量任务的自动化。

选项

`-a``--echo-all`

在读取行时向标准输出打印所有内容。这个选项在脚本处理时比交互模式时更有用。这个选项等效于设置变量 `ECHO` 为 `all` 。

`-A` `--no-align`

切换到非对齐输出模式。 （缺省输出模式是对齐的。）

`-c` `_command_` `--command=``_command_`

声明psql将执行一条查询字符串 `_command_` ，然后退出。 这在 shell 脚本里很有用。启动（设置）文件 (`psqlrc` 和 `~/.psqlrc`) 忽略这个选项。

`_command_` 必须是一条完全可以被服务器分析的字符串（也就是， 它不包含psql-特有的特性）， 或一个反斜杠命令。因此你不能混合使用 SQL和psql 元命令使用这个选项。 要想混合使用，你可以把字符串重定向到 psql里，像这样：
`echo '\x \\ SELECT * FROM foo;' | psql` 。 （ `\\` 用于隔开元命令)。

如果命令字符串包含多个 SQL 命令，那么他们将在一个事务里处理，除非在字符串里包含了明确的 `BEGIN` / `COMMIT` 命令把他们分成多个事务。这个和从 psql的标准输入里给它填充相同字符串不同。 此外，只有最后一个SQL命令的执行结果被返回。

-d _dbname_ --dbname=``_dbname_

指定想要连接的数据库名称。这相当于把 _dbname_ 作为命令行的第一个非选项参数。

如果该参数包含一个 = 标志或 开始于一个有效的URI前缀 (postgresql://
或 postgres://)， 它被视为一个 conninfo 字符串。 见Section 31.1.1获取更多信息。

-e --echo-queries

把所有发送给服务器的查询同时也回显到标准输出。 等效于把变量 ECHO 设置为 queries 。

-E --echo-hidden

回显由 \d 和其它反斜杠命令（内部命令）生成的实际查询。 你可以使用这个命令学习 psql
的内部操作。这等效于在psql里设置变量 ECHO_HIDDEN 。

-f _filename_ --file=``_filename_

使用 _filename_ 作为命令的语句源而不是交互式读入查询。psql 将在处理完文件后结束。
这个选项在很多方面等效于内部命令 \i 。

如果 _filename_ 是 - (连字符)， 则从标准输入读取。

使用这个选项与使用 psql < _filename_ 有微小的区别。 通常， 两者都回按照你预期那样
运行， 但是使用 -f 打开了一些很好的特性，比如带行号的错误信息。 而且， 使用这个选项
还可能减小启动的开销。 另一方面， 使用shell 输入重定向的方式(理论上)能保证生成和你手
工输入所有内容时收到的（输出）完全一样的输出。

-F _separator_ --field-separator=``_separator_

指定 _separator_ 作为不对齐输出的字段分隔符。 等效于 \pset fieldsep 或 \f 。

-h _hostname_ --host=``_hostname_

指定正在运行服务器的主机名。如果主机名以斜杠开头， 则它被用作到Unix域套接字的路
径。

-H --html

打开 HTML格式输出（模式）。等效于 \pset format html 或 \H 命令。

-l --list

列出所有可用的数据库，然后退出。 其他非连接选项将被忽略。这是类似于元命令 \list 。

-L _filename_ --log-file=``_filename_

除了正常的输出，把所有查询输出写到文件 _filename_ 中。

-n --no-readline

Do not use readline for line editing and do not use the history. This can be useful to turn off tab expansion when cutting and pasting.

`-o` `_filename_` `--output=``_filename_`

Put all query output into file `_filename_`. This is equivalent to the command `\o`.

`-p` `_port_` `--port=``_port_`

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

`-P` `_assignment_` `--pset=``_assignment_`

Specifies printing options, in the style of `\pset`. Note that here you have to separate name and value with an equal sign instead of a space. For example, to set the output format to LaTeX, you could write `-P format=latex`.

`-q` `--quiet`

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the `QUIET` variable to achieve the same effect.

`-R` `_separator_` `--record-separator=``_separator_`

Use `_separator_` as the record separator for unaligned output. This is equivalent to the `\pset recordsep` command.

`-s` `--single-step`

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

`-S` `--single-line`

Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.

> **Note:** This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

`-t` `--tuples-only`

Turn off printing of column names and result row count footers, etc. This is equivalent to the `\t` command.

`-T` `_table_options_` `--table-attr=``_table_options_`

Specifies options to be placed within the HTML `table` tag. See `\pset` for details.

`-U` `_username_` `--username=``_username_`

Connect to the database as the user `_username_` instead of the default. (You must have permission to do so, of course.)

`-v` `_assignment_` `--set=``_assignment_` `--variable=``_assignment_`

Perform a variable assignment, like the `\set` meta-command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To set a variable with an empty value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

`-V` `--version`

Print the psql version and exit.

`-w` `--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

`-W` `--password`

Force psql to prompt for a password before connecting to a database.

This option is never essential, since psql will automatically prompt for a password if the server demands password authentication. However, psql will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

`-x` `--expanded`

Turn on the expanded table formatting mode. This is equivalent to the `\x` command.

`-X,` `--no-psqlrc`

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

`-z` `--field-separator-zero`

Set the field separator for unaligned output to a zero byte.

`-0` `--record-separator-zero`

Set the record separator for unaligned output to a zero byte. This is useful for interfacing, for example, with `xargs -0`.

`-1` `--single-transaction`

When psql executes a script, adding this option wraps `BEGIN` / `COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

`-?` `--help`

Show help about psql command line arguments, and exit.

Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own occurs (e.g. out of memory, file not found), 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting to a Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server, and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all of these options are required; there are useful defaults. If you omit the host name, psql will connect via a Unix-domain socket to a

server on the local host, or via TCP/IP to `localhost` on machines that don't have Unix-domain sockets. The default port number is determined at compile time. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults aren't quite right, you can save yourself some typing by setting the environment variables `PGDATABASE` , `PGHOST` , `PGPORT` and/or `PGUSER` to appropriate values. (For additional environment variables, see Section 31.14.) It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. See Section 31.15 for more information.

An alternative way to specify connection parameters is in a `conninfo` string or a URI, which is used instead of a database name. This mechanism give you very wide control over the connection. For example:

```
$ <kbd class="literal">psql "service=myservice sslmode=require"</kbd>
$ <kbd class="literal">psql postgresql://dbmaster:5433/mydb?sslmode=require</kbd>
```

This way you can also use LDAP for connection parameter lookup as described in Section 31.17. See Section 31.1.2 for more information on all the available connection options.

If the connection could not be made for any reason (e.g., insufficient privileges, server is not running on the targeted host, etc.), psql will return an error and terminate.

If at least one of standard input or standard output are a terminal, then psql sets the client encoding to "auto", which will detect the appropriate client encoding from the locale settings (`LC_CTYPE` environment variable on Unix systems). If this doesn't work out as expected, the client encoding can be overridden using the environment variable `PGCLIENTENCODING` .

Entering SQL Commands

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string `=>` . For example:

```
$ <kbd class="literal">psql testdb</kbd>
psql (9.3.1)
Type "help" for help.

testdb=>
```

At the prompt, the user can type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the

command was sent and executed without error, the results of the command are displayed on the screen.

Whenever a command is executed, psql also polls for asynchronous notification events generated by LISTEN and NOTIFY.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands make psql more useful for administration or scripting. Meta-commands are often called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace in an argument you can quote it with single quotes. To include a single quote in an argument, write two single quotes within single-quoted text. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\b` (backspace), `\r` (carriage return), `\f` (form feed), `` \``_digits_ `` (octal), and `` \x``_digits_ `` (hexadecimal). A backslash preceding any other character within single-quoted text quotes that single character, whatever it is.

Within an argument, text that is enclosed in backquotes (```) is taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) replaces the backquoted text.

If an unquoted colon (`:`) followed by a psql variable name appears within an argument, it is replaced by the variable's value, as described in *SQL* Interpolation.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (`"`) protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz` , and `"A weird"" name"` becomes `A weird" name` .

Parsing for arguments stops at the end of the line, or when another unquoted backslash is found. An unquoted backslash is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

`\c` or `\connect` `[` `_dbname_` [`_username_`] [`_host_`] [`_port_`]]

Establishes a new connection to a PostgreSQL server. If the new connection is successfully made, the previous connection is closed. If any of `_dbname_`, `_username_`, `_host_` or `_port_` are omitted or specified as `-`, the value of that parameter from the previous connection is used. If there is no previous connection, the libpq default for the parameter's value is used.

If the connection attempt failed (wrong user name, access denied, etc.), the previous connection will only be kept if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

`\C [` `_title_`]

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title` `_title_`. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\cd [` `_directory_`]

Changes the current working directory to `_directory_`. Without argument, changes to the current user's home directory.

> **Tip:** To print your current working directory, use `\! pwd`.

`\conninfo`

Outputs information about the current database connection.

`\copy {` `_table_` [(`_column_list_`)] | (`_query_`) } { `from` | `to` } { `_'filename'_` | program `_'command'_` | stdin | stdout | pstdin | pstdout } [[with] (`_option_` [, ...])]

Performs a frontend (client) copy. This is an operation that runs an SQL COPY command, but instead of the server reading or writing the specified file, psql reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

When `program` is specified, `_command_` is executed by psql and the data from or to `_command_` is routed between the server and the client. This means that the execution privileges are those of the local user, not the server, and no SQL superuser privileges are required.

`\copy ... from stdin | to stdout` reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches EOF. Output is sent to the same place as command output. To read/write from psql's standard input or output, use `pstdin` or `pstdout`. This option is useful for populating tables in-line within a SQL script file.

The syntax of the command is similar to that of the SQL COPY command, and `_option_` must indicate one of the options of the SQL COPY command. Note that, because of this, special parsing rules apply to the `\copy` command. In particular, the variable substitution rules and backslash escapes do not apply.

> **Tip:** This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server connection. For large amounts of data the SQL command might be preferable.

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d[S+] [` `_pattern_`]

For each relation (table, view, index, sequence, or foreign table) or composite type matching the `_pattern_`, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults. Associated indexes, constraints, rules, and triggers are also shown. For foreign tables, the associated foreign server is shown as well. ("Matching the pattern" is defined in *Patterns* below.)

For some types of relation, `\d` shows additional information for each column: column values for sequences, indexed expression for indexes and foreign data wrapper options for foreign tables.

The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table, the view definition if the relation is a view.

By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

> **Note:** If `\d` is used without a `_pattern_` argument, it is equivalent to `\dtvsE` which will show a list of all visible tables, views, sequences and foreign tables. This is purely a convenience measure.

`\da[S] [` `_pattern_`]

Lists aggregate functions, together with their return type and the data types they operate on. If `_pattern_` is specified, only aggregates whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\db[+] [` `_pattern_`]

Lists tablespaces. If `_pattern_` is specified, only tablespaces whose names match the pattern are shown. If `+` is appended to the command name, each object is listed with its associated permissions.

`\dc[S+] [` `_pattern_`]

Lists conversions between character-set encodings. If `_pattern_` is specified, only conversions whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If `+` is appended to the command name, each object is listed with its associated description.

`\dC[+] [` `_pattern_`]

Lists type casts. If `_pattern_` is specified, only casts whose source or target types match the pattern are listed. If `+` is appended to the command name, each object is listed with its associated description.

`\dd[S] [` `_pattern_`]

Shows the descriptions of objects of type `constraint` , `operator class` , `operator family` , `rule` , and `trigger` . All other comments may be viewed by the respective backslash commands for those object types.

`\dd` displays descriptions for objects matching the `_pattern_` , or of visible objects of the appropriate type if no argument is given. But in either case, only objects that have a description are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

Descriptions for objects can be created with the COMMENT SQL command.

`\ddp [` `_pattern_`]

Lists default access privilege settings. An entry is shown for each role (and schema, if applicable) for which the default privilege settings have been changed from the built-in defaults. If `_pattern_` is specified, only entries whose role name or schema name matches the pattern are listed.

The ALTER DEFAULT PRIVILEGES command is used to set default access privileges. The meaning of the privilege display is explained under GRANT.

`\dD[S+] [` `_pattern_`]

Lists domains. If `_pattern_` is specified, only domains whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If `+` is appended to the command name, each object is listed with its associated permissions and description.

`\dE[S+] [` `_pattern_`] `\di[S+] [` `_pattern_`] `\dm[S+] [` `_pattern_`] `\ds[S+] [` `_pattern_`] `\dt[S+] [` `_pattern_`] `\dv[S+] [` `_pattern_`]

In this group of commands, the letters `E`, `i`, `m`, `s`, `t`, and `v` stand for foreign table, index, materialized view, sequence, table, and view, respectively. You can specify any or all of these letters, in any order, to obtain a listing of objects of these types. For example, `\dit` lists indexes and tables. If `+` is appended to the command name, each object is listed with its physical size on disk and its associated description, if any. If `_pattern_` is specified, only objects whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\des[+] [` `_pattern_`]

Lists foreign servers (mnemonic: "external servers"). If `_pattern_` is specified, only those servers whose name matches the pattern are listed. If the form `\des+` is used, a full description of each server is shown, including the server's ACL, type, version, options, and description.

`\det[+] [` `_pattern_`]

Lists foreign tables (mnemonic: "external tables"). If `_pattern_` is specified, only entries whose table name or schema name matches the pattern are listed. If the form `\det+` is used, generic options and the foreign table description are also displayed.

`\deu[+] [` `_pattern_`]

Lists user mappings (mnemonic: "external users"). If `_pattern_` is specified, only those mappings whose user names match the pattern are listed. If the form `\deu+` is used, additional information about each mapping is shown.

Caution

`\deu+` might also display the user name and password of the remote user, so care should be taken not to disclose them.

`\dew[+] [` `_pattern_`]

Lists foreign-data wrappers (mnemonic: "external wrappers"). If `_pattern_` is specified, only those foreign-data wrappers whose name matches the pattern are listed. If the form `\dew+` is used, the ACL, options, and description of the foreign-data wrapper are also shown.

`\df[antwS+] [` `_pattern_`]

Lists functions, together with their arguments, return types, and function types, which are classified as "agg" (aggregate), "normal", "trigger", or "window". To display only functions of specific type(s), add the corresponding letters `a` , `n` , `t` , or `w` to the command. If `_pattern_` is specified, only functions whose names match the pattern are shown. If the form `\df+` is used, additional information about each function, including security, volatility, language, source code and description, is shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

> **Tip:** To look up functions taking arguments or returning values of a specific type, use your pager's search capability to scroll through the `\df` output.

`\dF[+] [` `_pattern_`]

Lists text search configurations. If `_pattern_` is specified, only configurations whose names match the pattern are shown. If the form `\dF+` is used, a full description of each configuration is shown, including the underlying text search parser and the dictionary list for each parser token type.

`\dFd[+] [` `_pattern_`]

Lists text search dictionaries. If `_pattern_` is specified, only dictionaries whose names match the pattern are shown. If the form `\dFd+` is used, additional information is shown about each selected dictionary, including the underlying text search template and the option values.

`\dFp[+] [` `_pattern_`]

Lists text search parsers. If `_pattern_` is specified, only parsers whose names match the pattern are shown. If the form `\dFp+` is used, a full description of each parser is shown, including the underlying functions and the list of recognized token types.

`\dFt[+] [` `_pattern_`]

Lists text search templates. If `_pattern_` is specified, only templates whose names match the pattern are shown. If the form `\dFt+` is used, additional information is shown about each template, including the underlying function names.

`\dg[+] [` `_pattern_`]

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to `\du`.) If `_pattern_` is specified, only those roles whose names match the pattern are listed. If the form `\dg+` is used, additional information is shown about each role; currently this adds the comment for each role.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\dL[S+] [` `_pattern_`]

Lists procedural languages. If `_pattern_` is specified, only languages whose names match the pattern are listed. By default, only user-created languages are shown; supply the `S` modifier to include system objects. If `+` is appended to the command name, each language is listed with its call handler, validator, access privileges, and whether it is a system object.

`\dn[S+] [` `_pattern_`]

Lists schemas (namespaces). If `_pattern_` is specified, only schemas whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If `+` is appended to the command name, each object is listed with its associated permissions and description, if any.

`\do[S] [` `_pattern_`]

Lists operators with their operand and return types. If `_pattern_` is specified, only operators whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\dO[S+] [` `_pattern_`]

Lists collations. If `_pattern_` is specified, only collations whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If `+` is appended to the command name, each collation is listed with its associated description, if any. Note that only collations usable with the current database's encoding are shown, so the results may vary in different databases of the same installation.

`\dp [` `_pattern_`]

Lists tables, views and sequences with their associated access privileges. If `_pattern_` is specified, only tables, views and sequences whose names match the pattern are listed.

The GRANT and REVOKE commands are used to set access privileges. The meaning of the privilege display is explained under GRANT.

`\drds [` `_role-pattern_` [`_database-pattern_`]]

Lists defined configuration settings. These settings can be role-specific, database-specific, or both. `_role-pattern_` and `_database-pattern_` are used to select specific roles and databases to list, respectively. If omitted, or if `*` is specified, all settings are listed, including those not role-specific or database-specific, respectively.

The ALTER ROLE and ALTER DATABASE commands are used to define per-role and per-database configuration settings.

`\dT[S+] [` `_pattern_`]

Lists data types. If `_pattern_` is specified, only types whose names match the pattern are listed. If `+` is appended to the command name, each type is listed with its internal name and size, its allowed values if it is an `enum` type, and its associated permissions. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\du[+] [` `_pattern_`]

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to `\dg`.) If `_pattern_` is specified, only those roles whose names match the pattern are listed. If the form `\du+` is used, additional information is shown about each role; currently this adds the comment for each role.

`\dx[+] [` `_pattern_`]

Lists installed extensions. If `_pattern_` is specified, only those extensions whose names match the pattern are listed. If the form `\dx+` is used, all the objects belonging to each matching extension are listed.

`\dy[+] [` `_pattern_`]

Lists event triggers. If `_pattern_` is specified, only those event triggers whose names match the pattern are listed. If `+` is appended to the command name, each object is listed with its associated description.

`\e` or `\edit` `` [ ``_filename_ ``] [`_line_number_`]

If `_filename_` is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no `_filename_` is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means that if the query ends with (or contains) a semicolon, it is immediately executed. Otherwise it will merely wait in the query buffer; type semicolon or `\g` to send it, or `\r` to cancel.

If a line number is specified, psql will position the cursor on the specified line of the file or query buffer. Note that if a single all-digits argument is given, psql assumes it is a line number, not a file name.

> **Tip:** See under *Environment* for how to configure and customize your editor.

`\echo` `_text_` [...]

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=&gt; &lt;kbd class="literal"&gt;\echo `date`&lt;/kbd&gt;
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the trailing newline is not written.

> **Tip:** If you use the `\o` command to redirect your query output you might wish to use `\qecho` instead of this command.

`\ef [``_function_description_` [ `_line_number_`]]

This command fetches and edits the definition of the named function, in the form of a `CREATE OR REPLACE FUNCTION` command. Editing is done in the same way as for `\edit`. After the editor exits, the updated command waits in the query buffer; type semicolon or `\g` to send it, or `\r` to cancel.

The target function can be specified by name alone, or by name and arguments, for example `foo(integer, text)`. The argument types must be given if there is more than one function of the same name.

If no function is specified, a blank `CREATE FUNCTION` template is presented for editing.

If a line number is specified, psql will position the cursor on the specified line of the function body. (Note that the function body typically does not begin on the first line of the file.)

> **Tip:** See under *Environment* for how to configure and customize your editor.

`\encoding [` `_encoding_`]

Sets the client character set encoding. Without an argument, this command shows the current encoding.

`\f [` `_string_`]

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

`\g` [{ `_filename_` | `|``_command_` }]

Sends the current query input buffer to the server and optionally stores the query's output in `_filename_` or pipes the output into a separate Unix shell executing `_command_`. The file or command is written to only if the query successfully returns zero or more tuples, not if the query fails or is a non-data-returning SQL command.

A bare `\g` is essentially equivalent to a semicolon. A `\g` with argument is a "one-shot" alternative to the `\o` command.

`\gset` [`_prefix_`]

Sends the current query input buffer to the server and stores the query's output into psql variables (see *Variables*). The query to be executed must return exactly one row. Each column of the row is stored into a separate variable, named the same as the column. For example:

```
=&gt; &lt;kbd class="literal"&gt;SELECT 'hello' AS var1, 10 AS var2&lt;/kbd&gt;
-&gt; &lt;kbd class="literal"&gt;\gset&lt;/kbd&gt;
=&gt; &lt;kbd class="literal"&gt;\echo :var1 :var2&lt;/kbd&gt;
hello 10
```

If you specify a `_prefix_`, that string is prepended to the query's column names to create the variable names to use:

```
=&gt; &lt;kbd class="literal"&gt;SELECT 'hello' AS var1, 10 AS var2&lt;/kbd&gt;
-&gt; &lt;kbd class="literal"&gt;\gset result_&lt;/kbd&gt;
=&gt; &lt;kbd class="literal"&gt;\echo :result_var1 :result_var2&lt;/kbd&gt;
hello 10
```

If a column result is NULL, the corresponding variable is unset rather than being set.

If the query fails or does not return one row, no variables are changed.

`\h` or `\help` `[` `_command_`]

Gives syntax help on the specified SQL command. If `_command_` is not specified, then psql will list all the commands for which syntax help is available. If `_command_` is an asterisk (`*`), then syntax help on all SQL commands is shown.

> **Note:** To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type <kbd class="literal">\help alter table</kbd>.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` `_filename_`

Reads input from the file `_filename_` and executes it as though it had been typed on the keyboard.

> **Note:** If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\ir` `_filename_`

The `\ir` command is similar to `\i`, but resolves relative file names differently. When executing in interactive mode, the two commands behave identically. However, when invoked from a script, `\ir` interprets file names relative to the directory in which the script is located, rather than the current working directory.

`\l[+]` or `\list[+] [` `_pattern_`]

List the databases in the server and show their names, owners, character set encodings, and access privileges. If `_pattern_` is specified, only databases whose names match the pattern are listed. If `+` is appended to the command name, database sizes, default tablespaces, and descriptions are also displayed. (Size information is only available for databases that the current user can connect to.)

`\lo_export` `_loid_` `_filename_`

Reads the large object with OID `_loid_` from the database and writes it to `_filename_`. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\lo_import` `_filename_` [`_comment_`]

Stores the file into a PostgreSQL large object. Optionally, it associates the given comment with the object. Example:

```
foo=&gt; &lt;kbd class="literal"&gt;\lo_import '/home/peter/pictures/photo.xcf' 'a pictur
lo_import 152801
```

The response indicates that the large object received object ID 152801, which can be used to access the newly-created large object in the future. For the sake of readability, it is recommended to always associate a human-readable comment with every object. Both OIDs and comments can be viewed with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

`\lo_list`

Shows a list of all PostgreSQL large objects currently stored in the database, along with any comments provided for them.

`\lo_unlink` `_loid_`

Deletes the large object with OID `_loid_` from the database.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\o` [{ `_filename_` | `|``_command_` }]

Saves future query results to the file `_filename_` or pipes future results into a separate Unix shell to execute `_command_`. If no arguments are specified, the query output will be reset to the standard output.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

> **Tip:** To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\password [` `_username_`]

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

`\prompt [` `_text_`] `_name_`

Prompts the user to supply text, which is assigned to the variable `_name_`. An optional prompt string, `_text_`, can be specified. (For multiword prompts, surround the text with single quotes.)

By default, `\prompt` uses the terminal for input and output. However, if the `-f` command line switch was used, `\prompt` uses standard input and standard output.

`\pset` `_option_` [`_value_`]

This command sets options affecting the output of query result tables. `_option_` indicates which option is to be set. The semantics of `_value_` vary depending on the selected option. For some options, omitting `_value_` causes the option to be toggled or unset, as described under the particular option. If no such behavior is mentioned, then omitting `_value_` just results in the current setting being displayed.

Adjustable printing options are:

`border`

The `_value_` must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML format, this will translate directly into the `border=...` attribute; in the other formats only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense. `latex` and `latex-longtable` also support a `border` value of 3 which adds a dividing line between each row.

`columns`

Sets the target width for the `wrapped` format, and also the width limit for determining whether output is wide enough to require the pager or switch to the vertical display in expanded auto mode. Zero (the default) causes the target width to be controlled by the environment variable `COLUMNS`, or the detected screen width if `COLUMNS` is not set. In addition, if `columns` is zero then the `wrapped` format only affects screen output. If `columns` is nonzero then file and pipe output is wrapped to that width as well.

`expanded` (or `x`)

If `_value_` is specified it must be either `on` or `off`, which will enable or disable expanded mode, or `auto`. If `_value_` is omitted the command toggles between the on and off settings. When expanded mode is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode. In the auto setting, the expanded mode is used whenever the query output is wider than the screen, otherwise the regular mode is used. The auto setting is only effective in the aligned and wrapped formats. In other formats, it always behaves as if the expanded mode is off.

`fieldsep`

Specifies the field separator to be used in unaligned output format. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).

`fieldsep_zero`

Sets the field separator to use in unaligned output format to a zero byte.

`footer`

If `_value_` is specified it must be either `on` or `off` which will enable or disable display of the table footer (the `` (``_n_ `` rows) count). If `_value_` is omitted the command toggles footer display on or off.

`format`

Sets the output format to one of `unaligned`, `aligned`, `wrapped`, `html`, `latex` (uses `tabular`), `latex-longtable`, or `troff-ms`. Unique abbreviations are allowed. (That would mean one letter is enough.)

`unaligned` format writes all columns of a row on one line, separated by the currently active field separator. This is useful for creating output that might be intended to be read in by other programs (for example, tab-separated or comma-separated format).

`aligned` format is the standard, human-readable, nicely formatted text output; this is the default.

`wrapped` format is like `aligned` but wraps wide data values across lines to make the output fit in the target column width. The target width is determined as described under the `columns` option. Note that psql will not attempt to wrap column header titles; therefore, `wrapped` format behaves the same as `aligned` if the total width needed for column headers exceeds the target.

The `html`, `latex`, `latex-longtable`, and `troff-ms` formats put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! This might not be necessary in HTML, but in LaTeX you must have a complete document wrapper. `latex-longtable` also requires the LaTeX `longtable` and `booktabs` packages.

`linestyle`

Sets the border line drawing style to one of `ascii`, `old-ascii` or `unicode`. Unique abbreviations are allowed. (That would mean one letter is enough.) The default setting is `ascii`. This option only affects the `aligned` and `wrapped` output formats.

`ascii` style uses plain ASCII characters. Newlines in data are shown using a `+` symbol in the right-hand margin. When the `wrapped` format wraps data from one line to the next without a newline character, a dot (`.`) is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

`old-ascii` style uses plain ASCII characters, using the formatting style used in PostgreSQL 8.4 and earlier. Newlines in data are shown using a `:` symbol in place of the left-hand column separator. When the data is wrapped from one line to the next without a newline character, a `;` symbol is used in place of the left-hand column separator.

`unicode` style uses Unicode box-drawing characters. Newlines in data are shown using a carriage return symbol in the right-hand margin. When the data is wrapped from one line to the next without a newline character, an ellipsis symbol is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

When the `border` setting is greater than zero, this option also determines the characters with which the border lines are drawn. Plain ASCII characters work everywhere, but Unicode characters look nicer on displays that recognize them.

`null`

Sets the string to be printed in place of a null value. The default is to print nothing, which can easily be mistaken for an empty string. For example, one might prefer `\pset null '(null)'` .

`numericlocale`

If `_value_` is specified it must be either `on` or `off` which will enable or disable display of a locale-specific character to separate groups of digits to the left of the decimal marker. If `_value_` is omitted the command toggles between regular and locale-specific numeric output.

`pager`

Controls use of a pager program for query and psql help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used.

When the `pager` option is `off` , the pager program is not used. When the `pager` option is `on` , the pager is used when appropriate, i.e., when the output is to a terminal and will not fit on the screen. The `pager` option can also be set to `always` , which causes the pager to be used for all terminal output regardless of whether it fits on the screen. `\pset pager` without a `_value_` toggles pager use on and off.

`recordsep`

Specifies the record (line) separator to use in unaligned output format. The default is a newline character.

`recordsep_zero`

Sets the record separator to use in unaligned output format to a zero byte.

`tableattr` (or `T`)

In HTML format, this specifies attributes to be placed inside the `table` tag. This could for example be `cellpadding` or `bgcolor` . Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border` . If no `_value_` is given, the table attributes are unset.

In `latex-longtable` format, this controls the proportional width of each column containing a left-aligned data type. It is specified as a whitespace-separated list of values, e.g. `'0.2 0.2 0.6'` . Unspecified output columns use the last specified value.

`title`

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no `_value_` is given, the title is unset.

`tuples_only` (or `t`)

If `_value_` is specified it must be either `on` or `off` which will enable or disable tuples-only mode. If `_value_` is omitted the command toggles between regular and tuples-only output. Regular output includes extra information such as column headers, titles, and various footers. In tuples-only mode, only actual table data is shown.

Illustrations of how these different formats look can be seen in the *Examples* section.

> **Tip:** There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.
>
> **Note:** It is an error to call `\pset` without any arguments. In the future this case might show the current status of all printing options.

`\q` or `\quit`

Quits the psql program. In a script file, only execution of that script is terminated.

`\qecho` `_text_` [...]

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s [` `_filename_`]

Print or save the command line history to `_filename_`. If `_filename_` is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU Readline library.

`\set [` `_name_` [`_value_` [...]]]

Sets the psql variable `_name_` to `_value_`, or if more than one value is given, to the concatenation of all of them. If only one argument is given, the variable is set with an empty value. To unset a variable, use the `\unset` command.

`\set` without any arguments displays the names and values of all currently-set psql variables.

Valid variable names can contain letters, digits, and underscores. See the section *Variables* below for details. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

> **Note:** This command is unrelated to the SQL command SET.

`\setenv [` `_name_` [`_value_`]]

Sets the environment variable `_name_` to `_value_`, or if the `_value_` is not supplied, unsets the environment variable. Example:

```
testdb=&gt; &lt;kbd class="literal"&gt;\setenv PAGER less&lt;/kbd&gt;
testdb=&gt; &lt;kbd class="literal"&gt;\setenv LESS -imx4F&lt;/kbd&gt;
```

`\sf[+]` `_function_description_`

This command fetches and shows the definition of the named function, in the form of a `CREATE OR REPLACE FUNCTION` command. The definition is printed to the current query output channel, as set by `\o`.

The target function can be specified by name alone, or by name and arguments, for example `foo(integer, text)`. The argument types must be given if there is more than one function of the same name.

If `+` is appended to the command name, then the output lines are numbered, with the first line of the function body being line 1.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` `_table_options_`

Specifies attributes to be placed within the `table` tag in HTML output format. This command is equivalent to `\pset tableattr` `_table_options_`.

`\timing [` `_on_` | `_off_`]

Without parameter, toggles a display of how long each SQL statement takes, in milliseconds. With parameter, sets same.

`\unset` `_name_`

Unsets (deletes) the psql variable `_name_`.

`\w` `_filename_` `\w` ``|``_command_``

Outputs the current query buffer to the file `_filename_` or pipes it to the Unix command `_command_`.

`\watch [` `_seconds_`]

Repeatedly execute the current query buffer (like `\g`) until interrupted or the query fails. Wait the specified number of seconds (default 2) between executions.

`\x [` `_on_` | `_off_` | `_auto_`]

Sets or toggles expanded table formatting mode. As such it is equivalent to `\pset expanded` .

`\z [` `_pattern_`]

Lists tables, views and sequences with their associated access privileges. If a `_pattern_` is specified, only tables, views and sequences whose names match the pattern are listed.

This is an alias for `\dp` ("display privileges").

`\! [` `_command_`]

Escapes to a separate Unix shell or executes the Unix command `_command_` . The arguments are not further interpreted; the shell will see them as-is. In particular, the variable substitution rules and backslash escapes do not apply.

`\?`

Shows help information about the backslash commands.

Patterns

The various `\d` commands accept a `_pattern_` parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo` . As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO""BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar` .

Whenever the `_pattern_` parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path — this is equivalent to using `*` as the pattern. (An object is said to be *visible* if its containing schema is in the search path and no object of the same kind and name appears earlier in the search path. This is equivalent to the statement that the object can be referenced by name without explicit schema qualification.) To see all objects in the database regardless of visibility, use `*.*` as the pattern.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, `\dt int*` displays tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.*bar*` displays all tables whose table name includes `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations such as character classes, for example `[0-9]` to match any digit. All regular expression special characters work as specified in Section 9.7.3, except for `.` which is taken as a separator as mentioned above, `*` which is translated to the regular-expression notation `.*`, `?` which is translated to `.`, and `$` which is matched literally. You can emulate these pattern characters at need by writing `?` for `.`, `(``_R_`+|) for `_R_`*, or* `(``_R_`|) *for* `_R_`*?.* `$` *is not needed as a regular-expression character since the pattern must match the whole name, unlike the usual interpretation of regular expressions (in other words,* `$` *is automatically appended to your pattern). Write*
` `` at the beginning and/or end if you don't wish the pattern to be anchored. Note that withir ``
\do`).

Advanced Features

Variables

psql provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. The name must consist of letters (including non-Latin letters), digits, and underscores.

To set a variable, use the psql meta-command `\set`. For example,

```
testdb=> <kbd class="literal">\set foo bar</kbd>
```

sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon, for example:

```
testdb=> <kbd class="literal">\echo :foo</kbd>
bar
```

This works in both regular SQL commands and meta-commands; there is more detail in *SQL Interpolation*, below.

If you call `\set` without a second argument, the variable is set, with an empty string as value. To unset (i.e., delete) a variable, use the command `\unset` . To show the values of all variables, call `\set` without any argument.

> **Note:** The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

A number of these variables are treated specially by psql. They represent certain option settings that can be changed at run time by altering the value of the variable, or in some cases represent changeable state of psql. Although you can use these variables for other purposes, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables' names consist of all upper-case ASCII letters (and possibly digits and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables follows.

`AUTOCOMMIT`

When `on` (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When `off` or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END` . The autocommit-off mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

> **Note:** In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK` . Also keep in mind that if you exit the session without committing, your work will be lost.
>
> **Note:** The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you might wish to set it in the system-wide `psqlrc` file or your `~/.psqlrc` file.

`COMP_KEYWORD_CASE`

Determines which letter case to use when completing an SQL key word. If set to `lower` or `upper` , the completed word will be in lower or upper case, respectively. If set to `preserve-lower` or `preserve-upper` (the default), the completed word will be in the case of

the word already entered, but words being completed without anything entered will be in lower or upper case, respectively.

`DBNAME`

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`ECHO`

If set to `all`, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or executed. To select this behavior on program start-up, use the switch `-a`. If set to `queries`, psql merely prints all queries as they are sent to the server. The switch for this is `-e`.

`ECHO_HIDDEN`

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

`ENCODING`

The current client character set encoding.

`FETCH_COUNT`

If this variable is set to an integer value > 0, the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query might fail after having already displayed some rows.

> **Tip:** Although you can use any output format with this feature, the default `aligned` format tends to look bad because each group of `FETCH_COUNT` rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HISTFILE`

The file name that will be used to store the history list. The default value is `~/.psql_history`. For example, putting:

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in `~/.psqlrc` will cause psql to maintain a separate history for each database.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

> **Note:** This feature was shamelessly plagiarized from Bash.

`HOST`

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`IGNOREEOF`

If unset, sending an EOF character (usually **Control**+**D**) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

> **Note:** This feature was shamelessly plagiarized from Bash.

`LASTOID`

The value of the last affected OID, as returned from an `INSERT` or `\lo_import` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

`ON_ERROR_ROLLBACK`

When `on`, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When `interactive`, such errors are only ignored in interactive sessions, and not when reading script files. When `off` (the default), a statement in a transaction block that generates an error aborts the entire transaction. The on_error_rollback-on mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

`ON_ERROR_STOP`

By default, command processing continues after an error. When this variable is set, it will instead stop immediately. In interactive mode, psql will return to the command prompt; otherwise, psql will exit, returning error code 3 to distinguish this case from fatal error conditions, which are reported using error code 1. In either case, any currently running scripts (the top-level script, if any, and any other scripts which it may have in invoked) will be terminated immediately. If the top-level command string contained multiple SQL commands, processing will stop with the current command.

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1` `PROMPT2` `PROMPT3`

These specify what the prompts psql issues should look like. See *Prompting* below.

`QUIET`

This variable is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

`SINGLELINE`

This variable is equivalent to the command line option `-S`.

`SINGLESTEP`

This variable is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

`VERBOSITY`

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

A key feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements, as well as the arguments of meta-commands. Furthermore, psql provides facilities for ensuring that variable values used as SQL literals and identifiers are properly quoted. The syntax for interpolating a value without any quoting is to prepend the variable name with a colon (`:`). For example,

```
testdb=> <kbd class="literal">\set foo 'my_table'</kbd>
testdb=> <kbd class="literal">SELECT * FROM :foo;</kbd>
```

would query the table `my_table`. Note that this may be unsafe: the value of the variable is copied literally, so it can contain unbalanced quotes, or even backslash commands. You must make sure that it makes sense where you put it.

When a value is to be used as an SQL literal or identifier, it is safest to arrange for it to be quoted. To quote the value of a variable as an SQL literal, write a colon followed by the variable name in single quotes. To quote the value as an SQL identifier, write a colon followed by the variable name in double quotes. These constructs deal correctly with quotes and other special characters embedded within the variable value. The previous example would be more safely written this way:

```
testdb=> <kbd class="literal">\set foo 'my_table'</kbd>
testdb=> <kbd class="literal">SELECT * FROM :"foo";</kbd>
```

Variable interpolation will not be performed within quoted SQL literals and identifiers. Therefore, a construction such as `':foo'` doesn't work to produce a quoted literal from a variable's value (and it would be unsafe if it did work, since it wouldn't correctly handle quotes embedded in the value).

One example use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then interpolate the variable's value as a quoted string:

```
testdb=> <kbd class="literal">\set content `cat my_file.txt`</kbd>
testdb=> <kbd class="literal">INSERT INTO my_table VALUES (:'content');</kbd>
```

(Note that this still won't work if `my_file.txt` contains NUL bytes. psql does not support embedded NUL bytes in variable values.)

Since colons can legally appear in SQL commands, an apparent attempt at interpolation (that is, `:name`, `:'name'`, or `:"name"`) is not replaced unless the named variable is currently set. In any case, you can escape a colon with a backslash to protect it from substitution.

The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntaxes for array slices and type casts are PostgreSQL extensions, which can sometimes conflict with the standard usage. The colon-quote syntax for escaping a variable's value as an SQL literal or identifier is a psql extension.

Prompting

The prompts psql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (`%`) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or [local:``_/dir/name_], if the Unix domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is `~` (tilde) if the database is your default database.

`%#`

If the session user is a database superuser, then a `#`, otherwise a `>`. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%R`

In prompt 1 normally `=`, but `^` if in single-line mode, and `!` if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by `-`, `*`, a single quote, a double quote, or a dollar sign, depending on whether psql expects more input because the command wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn't produce anything.

`%x`

Transaction status: an empty string when not in a transaction block, or `*` when in a transaction block, or `!` when in a failed transaction block, or `?` when the transaction state is indeterminate (for example, because there is no connection).

``%``_digits_``

The character with the indicated octal code is substituted.

``%:``_name_``:``

The value of the psql variable `_name_`. See the section *Variables* for details.

`%```*command*```

The output of `_command_`, similar to ordinary "back-tick" substitution.

`%[` ... `%]`

Prompts can contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%]`. Multiple pairs of these can occur within the prompt. For example:

```
testdb=&gt; \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

results in a boldfaced (`1;`) yellow-on-black (`33;40`) prompt on VT100-compatible, color-capable terminals.

To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

> **Note:** This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql supports the Readline library for convenient line editing and retrieval. The command history is automatically saved when psql exits and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. The queries generated by tab-completion can also interfere with other SQL commands, e.g. `SET TRANSACTION ISOLATION LEVEL`. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a Readline feature. Read its documentation for further details.)

Environment

`COLUMNS`

If `\pset columns` is zero, controls the width for the `wrapped` format and width for determining if wide output requires the pager or should be switched to the vertical format in expanded auto mode.

`PAGER`

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. The use of the pager can be disabled by using the `\pset` command.

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

Default connection parameters (see Section 31.14).

`PSQL_EDITOR` `EDITOR` `VISUAL`

Editor used by the `\e` and `\ef` commands. The variables are examined in the order listed; the first that is set is used.

The built-in default editors are `vi` on Unix systems and `notepad.exe` on Windows systems.

`PSQL_EDITOR_LINENUMBER_ARG`

When `\e` or `\ef` is used with a line number argument, this variable specifies the command-line argument used to pass the starting line number to the user's editor. For editors such as Emacs or vi, this is a plus sign. Include a trailing space in the value of the variable if there needs to be space between the option name and the line number. Examples:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

The default is `+` on Unix systems (corresponding to the default editor `vi`, and useful for many other common editors); but there is no default on Windows systems.

`PSQL_HISTORY`

Alternative location for the command history file. Tilde (`~`) expansion is performed.

`PSQLRC`

Alternative location of the user's `.psqlrc` file. Tilde (`~`) expansion is performed.

`SHELL`

Command executed by the `\!` command.

`TMPDIR`

Directory for storing temporary files. The default is `/tmp` .

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Files

- Unless it is passed an `-X` or `-c` option, psql attempts to read and execute commands from the system-wide `psqlrc` file and the user's `~/.psqlrc` file before starting up. (On Windows, the user's startup file is named `%APPDATA%\postgresql\psqlrc.conf` .) See `_PREFIX_` /share/psqlrc.sample for information on setting up the system-wide file. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).

  The location of the user's `~/.psqlrc` file can also be set explicitly via the `PSQLRC` environment setting.

- Both the system-wide `psqlrc` file and the user's `~/.psqlrc` file can be made psql-version-specific by appending a dash and the PostgreSQL major or minor psql release number, for example `~/.psqlrc-9.2` or `~/.psqlrc-9.2.5` . The most specific version-matching file will be read in preference to a non-version-specific file.

- The command-line history is stored in the file `~/.psql_history` , or `%APPDATA%\postgresql\psql_history` on Windows.

  The location of the history file can also be set explicitly via the `PSQL_HISTORY` environment setting.

Notes

- In an earlier life psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. As of PostgreSQL 8.4 this is no longer allowed.
- psql works best with servers of the same or an older major version. Backslash commands are particularly likely to fail if the server is of a newer version than psql itself. However, backslash commands of the `\d` family should work with servers of versions back to 7.4, though not necessarily with servers newer than psql itself. The general functionality of running SQL commands and displaying query results should also work with servers of a newer major version, but this cannot be guaranteed in all cases.

  If you want to use psql to connect to several servers of different major versions, it is recommended that you use the newest version of psql. Alternatively, you can keep a copy of psql from each major version around and be sure to use the version that matches the respective server. But in practice, this additional complication should not be necessary.

Notes for Windows Users

psql is built as a "console application". Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within psql. If psql detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

- Set the code page by entering <kbd class="literal">cmd.exe /c chcp 1252</kbd>. (1252 is a code page that is appropriate for German; replace it with your value.) If you are using Cygwin, you can put this command in `/etc/profile`.
- Set the console font to `Lucida Console`, because the raster font does not work with the ANSI code page.

Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
testdb=> <kbd class="literal">CREATE TABLE my_table (</kbd>
testdb(> <kbd class="literal"> first integer not null default 0,</kbd>
testdb(> <kbd class="literal"> second text)</kbd>
testdb-> <kbd class="literal">;</kbd>
CREATE TABLE
```

Now look at the table definition again:

```
testdb=> <kbd class="literal">\d my_table</kbd>
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Now we change the prompt to something more interesting:

```
testdb=> <kbd class="literal">\set PROMPT1 '%n@%m %~%R%# '</kbd>
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can display tables in different ways by using the `\pset` command:

```
peter@localhost testdb=> <kbd class="literal">\pset border 2</kbd>
Border style is 2.
peter@localhost testdb=> <kbd class="literal">SELECT * FROM my_table;</kbd>
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> <kbd class="literal">\pset border 0</kbd>
Border style is 0.
peter@localhost testdb=> <kbd class="literal">SELECT * FROM my_table;</kbd>
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> <kbd class="literal">\pset border 1</kbd>
Border style is 1.
peter@localhost testdb=> <kbd class="literal">\pset format unaligned</kbd>
Output format is unaligned.
peter@localhost testdb=> <kbd class="literal">\pset fieldsep ","</kbd>
Field separator is ",".
peter@localhost testdb=> <kbd class="literal">\pset tuples_only</kbd>
Showing only tuples.
peter@localhost testdb=> <kbd class="literal">SELECT second, first FROM my_table;</kbd>
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> <kbd class="literal">\a \t \x</kbd>
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> <kbd class="literal">SELECT * FROM my_table;</kbd>
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

reindexdb

Name

reindexdb -- 重建PostgreSQL数据库索引

Synopsis

`reindexdb` [`_connection-option_` ...] [`--table` | `-t` `_table_`] ... [`--index` | `-i` `_index_`] ... [`_dbname_`]

`reindexdb` [`_connection-option_` ...] `--all` | `-a`

`reindexdb` [`_connection-option_` ...] `--system` | `-s` [`_dbname_`]

描述

reindexdb是一个重建数据库索引的工具。

reindexdb是 SQL 命令REINDEX的包装。 因此，用哪种方法重建索引都一样。

选项

reindexdb accepts the following command-line arguments: reindexdb接受下列命令行参数：

`-a``--all`

对所有数据库重建索引。

`[-d]` `_dbname_` `[--dbname=]``_dbname_`

指定要重建索引的数据库的名字。如果没有指定并且也没有使用 `-a` （或 `--all` ），那么数据库名从环境变量 `PGDATABASE` 中读取。如果没有设置环境变量，那么就使用连接使用的用户名。

`-e` `--echo`

回显reindexdb生成并发送给数据库的命令。

`-i` `_index_` `--index=``_index_`

仅对 `_index_` 索引进行重建。 可以通过写多个 `-i` 选项重建多个索引。

`-q` `--quiet`

不显示进程信息。

`-s` `--system`

对数据库的系统表重建索引。

`-t` `_table_` `--table=``_table_`

仅对 `_table_` 表重建索引。 可以通过写多个 `-t` 选项给多个表重建索引。

`-V` `--version`

打印reindexdb的版本然后退出。

`-?` `--help`

显示关于reindexdb命令行参数的帮助然后退出。

reindexdb还接受下列命令行参数作为连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头，则被用作到 Unix 域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器正在侦听的 TCP 端口或本地 Unix 域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制reindexdb在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则reindexdb 自动提示一个密码。不过，reindexdb 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

`--maintenance-db=``_dbname_`

指定要连接到的数据库的名字以发现其他应该重建索引的数据库。如果没有指定， 那么将使用 `postgres` 数据库，如果该数据库不存在， 则使用 `template1` 。

环境变量

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

缺省连接参数

这个功用，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。

诊断

如果遇到麻烦，参阅REINDEX和psql 获取潜在问题和错误信息的论述。数据库服务器必须在目标主机上运行。同样， 任何libpq前端库可获得的缺省设置和环境变量都将生效。

注意

reindexdb可能需要多次连接PostgreSQL 服务器，且每次都询问密码。此时使用 `~/.pgpass` 文件将会很方便。 参见Section 31.15获取更多信息。

例子

重建数据库 `test` 中的所有索引：

```
<samp class="literal">$</samp> <kbd class="literal">reindexdb test</kbd>
```

重建数据库 `abcd` 中名为 `foo` 的表上的 `bar` 索引：

```
<samp class="literal">$</samp> <kbd class="literal">reindexdb --table foo --index bar abc
```

又见

REINDEX

vacuumdb

Name

vacuumdb -- 收集垃圾并分析一个PostgreSQL数据库

Synopsis

`vacuumdb` [`_connection-option_` ...] [`_option_` ...] [`--table` | `-t` `_table_` [(`_column_` [,...])]] ... [`_dbname_`]

`vacuumdb` [`_connection-option_` ...] [`_option_` ...] `--all` | `-a`

描述

vacuumdb是一个用于整理PostgreSQL 数据库的工具。vacuumdb还将会生成用于PostgreSQL查询优化器的内部统计数据。

vacuumdb是 SQL 命令VACUUM的封装。 因此，用哪种方法清理数据库都一样。

选项

vacuumdb接受下列命令行参数：

`-a``--all`

清理所有数据库。

`[-d]` `_dbname_` `[--dbname=]``_dbname_`

声明要被清理或分析的数据库名称。如果没有声明这个参数并且没有使用 `-a` 或 `--all` ，那么从将环境变量 `PGDATABASE` 里读取数据库名。 如果那个也没有设置，则使用连接的用户名。

`-e` `--echo`

回显vacuumdb生成并发送给服务器的命令。

`-f` `--full`

执行"完全"清理。

`-F` `--freeze`

强制"冻结"元组。

`-q` `--quiet`

不显示进程信息。

`-t` `_table_` [(`_column_` [,...])] `--table=``_table_` [ ( `_column_` [,...])]

只是清理或分析 `_table_` 。 字段名称只是在与 `--analyze` 或 `--analyze-only` 选项联合使用时才需要声明。可以通过写多个 `-t` 选项清理多个表。

> **Tip:** 如果你声明了要清理的字段，你可能不得不在 shell 上逃逸圆括弧(见下面的例子)。

`-v` `--verbose`

在处理过程中打印详细信息。

`-V` `--version`

打印vacuumdb版本然后退出。

`-z` `--analyze`

计算用于优化器的统计数据。

`-Z` `--analyze-only`

只计算用于优化器的该数据库的统计值（不清理）。

`-?` `--help`

显示关于vacuumdb命令行参数的帮助然后退出。

vacuumdb还接受下列命令行参数作为连接参数：

`-h` `_host_` `--host=``_host_`

指定运行服务器的主机名。如果数值以斜杠开头，则被用作到 Unix 域套接字的路径。

`-p` `_port_` `--port=``_port_`

指定服务器正在侦听的 TCP 端口或本地 Unix 域套接字文件的扩展(描述符)。

`-U` `_username_` `--username=``_username_`

连接的用户名。

`-w` `--no-password`

从不发出密码提示问题。如果服务器要求密码认证并且密码不可用于其他意思如 `.pgpass` 文件，则连接尝试将会失败。 该选项在批量工作和不存在用户输入密码的脚本中很有帮助。

`-W` `--password`

强制vacuumdb在连接到数据库之前提示一个密码。

这个选项从来不是至关重要的，因为如果服务器需求密码认证，则vacuumdb 自动提示一个密码。不过，vacuumdb 将在找出服务器想要一个密码上浪费一个连接尝试。在某些情况下，值得输入 `-W` 以避免额外的连接尝试。

`--maintenance-db=``_dbname_`

指定要连接到的数据库的名字以发现其他应该清理的数据库。如果没有指定， 那么将使用 `postgres` 数据库，如果该数据库不存在， 则使用 `template1` 。

环境变量

`PGDATABASE` `PGHOST` `PGPORT` `PGUSER`

缺省连接参数

这个工具，类似大多数其他PostgreSQL实用工具， 也使用由libpq支持的环境变量（参阅Section 31.14）。

诊断

如果出差错了。参阅VACUUM和psql 获取关于错误信息和可能问题的详细描述。数据库服务器必须在目标主机上运行。 同样，任何libpq前端库可获得的缺省设置和环境变量都将生效。

注意

vacuumdb可能需要与PostgreSQL 服务器连接若干次，每次都询问口令。在这种情况下，设立一个 `~/.pgpass` 文件是比较方便的。参阅Section 31.15获取更多信息。

例子

整理数据库 `test` ：

```
<samp class="literal">$</samp> <kbd class="literal">vacuumdb test</kbd>
```

为优化器清理和分析一个名为 `bigdb` 的数据库：

```
<samp class="literal">$</samp> <kbd class="literal">vacuumdb --analyze bigdb</kbd>
```

清理数据库 `xyzzy` 中名为 `foo` 的表，并且为优化器分析列 `bar` ：

```
<samp class="literal">$</samp> <kbd class="literal">vacuumdb --analyze --verbose --table
```

又见

VACUUM

III. PostgreSQL 服务器应用程序

这部分包括PostgreSQL服务器应用和支持工具的参考信息。 这些命令只能用于在数据库服务器所在的主机上运行。 其它工具程序在Reference II, *PostgreSQL 客户端应用程序*中列出。

Table of Contents

initdb

Name

initdb -- 创建一个新的PostgreSQL数据库簇（cluster）

Synopsis

`initdb` [`_option_` ...] [`--pgdata` | `-D`] `_directory_`

描述

`initdb` 创建一个新的 PostgreSQL数据库簇（cluster）。 一个数据库簇（cluster）是由单个服务端实例管理的多个数据库的集合。

创建数据库系统包括创建数据库数据的宿主目录，生成共享的系统表 （不属于任何特定数据库的表）和创建 `template1` 和 `postgres` 数据库。 当你以后再创建一个新数据库时， `template1` 数据库里所有内容都会拷贝过来。 （因此，任何在 `template1` 里面安装的东西都自动拷贝到以后创建的数据库中。） `postgres` 数据库是一个缺省数据库，用于给用户、工具、第三方应用提供缺省数据库。

尽管 `initdb` 会尝试创建相应的数据目录， 但经常会没有权限做这件事。因为所要创建目录的父目录通常被 root所拥有。要初始化这种设置，用root创建一个空数据目录，然后用 `chown` 把该目录的所有权交给数据库用户帐号，然后 `su` 成数据库用户，最后以数据库用户身份运行 `initdb` 。

`initdb` 必须以运行数据库服务端的用户身份运行， 因为服务端需要访问 `initdb` 创建的目录和文件。 因为服务端通常是以非root身份运行的，因此一般也就不以root用户运行 `initdb` （事实上将拒绝你以root用户运行它）。

`initdb` 初始化该数据库簇（cluster）的缺省区域和字符集编码。 字符编码排序(`LC_COLLATE`)和字符集类 (`LC_CTYPE` , 也就是大写、小写、数字等)可以在数据库创建的时候独立设置。 `initdb` 决定了 `template1` 数据库， 这个设置将成为所有其它（以后新建）数据库的缺省。

要更改默认的排序顺序或字符集分类，使用 `--lc-collate` 和 `--lc-ctype` 选项。 使用 `C` 或 `POSIX` 之外的字符编码排序还会有性能影响。 因此在运行 `initdb` 的时候就做出正确的选择是非常重要的。

在服务端启动时，你可以使用 `--locale` 设置缺省的所有语言环境类别，包括排序顺序和字符集种类。 其余的语言环境类别以后可以修改。所有服务器区域值(`lc_*`) 可以用 `SHOW ALL` 显示。 更多细节可以在Section 22.1找到。

要修改缺省编码，可以使用 `--encoding` 选项。 更多细节可以在Section 22.3找到。

选项

`-A` `_authmethod_``--auth=``_authmethod_`

这个选项指定本地用户在 `pg_hba.conf` 里面设置的认证方法。 (`host` 和 `local` 所在行)。 除非你信任所有本地系统用户。 否则不要使用 `trust` 。 `trust` 是缺省的。

`--auth-host=``_authmethod_`

这个选项指定本地TCP/IP联接用户在 `pg_hba.conf` 里面设置的认证方法。 (`host` 所在行)。

`--auth-local=``_authmethod_`

这个选项指定本地Unix-domain socket联接用户在 `pg_hba.conf` 里面设置的认证方法。 (`local` 所在行)。

`-D` `_directory_` `--pgdata=``_directory_`

这个选项声明数据库簇（cluster）应该存放在哪个目录。这是 `initdb` 惟一（必须）需要的信息，但是你可以通过设置 `PGDATA` 环境变量来避免键入，这样做可能方便一些，因为稍后数据库服务器 (`postgres`) 可以通过同一个变量找到数据库目录。

`-E` `_encoding_` `--encoding=``_encoding_`

选择模板数据库的编码方式。这将是你以后创建的数据库的缺省编码方式， 除非你创建数据库时覆盖了它。缺省是从区域设置中获得的，如果没有区域设置，就是 `SQL_ASCII` 。 PostgreSQL 服务器支持的字符集在Section 22.3.1里描述。

`-k` `--data-checksums`

使用数据页产生校验和，以帮助I/O系统，否则将静音检测损坏。 启用校验和可能产生明显的性能损失。 此选项只能在初始化过程中设置，并且以后不能更改。 如果设置，那么在所有数据库中计算所有对象的校验和。

`--locale=``_locale_`

为数据库簇（cluster）设置缺省的区域。如果没有指定这个选项， 那么区域是从 `initdb` 运行的环境中继承过来的。 区域设置在Section 22.1里描述。

`--lc-collate=``_locale_` `--lc-ctype=``_locale_` `--lc-messages=``_locale_` `--lc-monetary=``_locale_` `--lc-numeric=``_locale_` `--lc-time=``_locale_`

类似 `--locale` ，但是只设置特殊范畴的区域。

`--no-locale`

相当于 `--locale=C` 。

`-N` `--nosync`

缺省情况下 `initdb` 会等待所有的文件被安全地写入到磁盘中。 此选项会导致 `initdb` 不进行等待，这样更快， 但意味着随后的操作系统崩溃可能把数据目录损坏。

`--pwfile=``_filename_`

令 `initdb` 从文件中读取数据库超级用户的密码。该文件的第一行被作为密码。

`-S` `--sync-only`

安全地写入所有数据库文件到磁盘然后退出。这并不进行任何普通initdb操作。

`-T` `_CFG_` `--text-search-config=``_CFG_`

设置缺省的文本搜索配置。 见default_text_search_config了解更多信息。

`-U` `_username_` `--username=``_username_`

选择数据库超级用户的用户名。缺省是运行 `initdb` 的用户的有效用户。超级用户的名字是什么并不重要， 但是可以选择习惯的名字postgres，即使和操作系统的用户名不一样也没关系。

`-W` `--pwprompt`

令 `initdb` 提示输入数据库超级用户的口令。 如果你不准备使用口令认证，这个选项并不重要。否则你将不能使用口令认证，直到你设置了口令。

`-X` `_directory_` `--xlogdir=``_directory_`

此选项指定事务日志应该存放在哪个目录。

其它不常用的参数还有：

`-d` `--debug`

从初始化后端打印调试输出以及一些其它的一些普通用户不太感兴趣的信息。 初始化后端是 `initdb` 用于创建系统表的程序。这个选项生成大量非常枯燥的输出。

`-L` `_directory_`

告诉 `initdb` 初始化数据库时所需要的输入文件的位置。 通常不需要。如果需要你明确指定的话，程序会提示你输入。

`-n` `--noclean`

缺省时，当 `initdb` 创建数据库簇（cluster）时出错， 它将在检测到不能结束工作之前将其创建的所有文件删除。 这个选项禁止任何清理动作，因而对调试很有用。

其它参数：

`-V` `--version`

输出initdb命令的版本信息，然后退出。

`-?` `--help`

显示initdb命令的帮助信息，然后退出。

环境变量

`PGDATA`

指定数据库簇（cluster）存储的目录; 可以使用 `-D` 选项覆盖。

此实用工具，像大多其他的PostgreSQL 实用工具， 还使用libpq支持的环境变量libpq (见 Section 31.14)。

注意

`initdb` 还可以通过调用 `pg_ctl initdb` .

另请参阅

pg_ctl, postgres

pg_controldata

Name

pg_controldata -- 显示一个集群的控制信息

Synopsis

`pg_controldata` [`_option_`] [`_datadir_`]

描述

`pg_controldata` 打印那些在 `initdb` 过程中初始化的信息， 比如表版本。它还显示有关预写日志和检查点处理相关的信息。这些信息是集群范围内有效的， 并不和某个数据库相关。

这个命令只应该由安装服务器的用户运行，因为它要求对数据目录的读访问权限。 你可以在命令行上声明数据目录，或者使用 `PGDATA` 环境变量。 这个工具支持 `-V` 和 `--version` 选项，可以打印 pg_controldata的版本然后退出。还支持 `-?` 和 `--help` 选项，输出支持的参数。

环境变量

`PGDATA`

缺省数据目录位置

pg_ctl

Name

pg_ctl -- initialize, start, stop, or control a PostgreSQL server

Synopsis

`pg_ctl` `init[db]` [`-s`] [`-D` `_datadir_`] [`-o` `_initdb-options_`]

`pg_ctl` `start` [`-w`] [`-t` `_seconds_`] [`-s`] [`-D` `_datadir_`] [`-l` `_filename_`] [`-o` `_options_`] [`-p` `_path_`] [`-c`]

`pg_ctl` `stop` [`-W`] [`-t` `_seconds_`] [`-s`] [`-D` `_datadir_`] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`]

`pg_ctl` `restart` [`-w`] [`-t` `_seconds_`] [`-s`] [`-D` `_datadir_`] [`-c`] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-o` `_options_`]

`pg_ctl` `reload` [`-s`] [`-D` `_datadir_`]

`pg_ctl` `status` [`-D` `_datadir_`]

`pg_ctl` `promote` [`-s`] [`-D` `_datadir_`]

`pg_ctl` `kill` `_signal_name_` `_process_id_`

`pg_ctl` `register` [`-N` `_servicename_`] [`-U` `_username_`] [`-P` `_password_`] [`-D` `_datadir_`] [`-S` `a[uto]` | `d[emand]`] [`-w`] [`-t` `_seconds_`] [`-s`] [`-o` `_options_`]

`pg_ctl` `unregister` [`-N` `_servicename_`]

Description

pg_ctl is a utility for initializing a PostgreSQL database cluster, starting, stopping, or restarting the PostgreSQL database server (postgres), or displaying the status of a running server. Although the server can be started manually, pg_ctl encapsulates tasks such as redirecting log output and properly detaching from the terminal and process group. It also provides convenient options for controlled shutdown.

The `init` or `initdb` mode creates a new PostgreSQL database cluster. A database cluster is a collection of databases that are managed by a single server instance. This mode invokes the `initdb` command. See initdb for details.

In `start` mode, a new server is launched. The server is started in the background, and its standard input is attached to `/dev/null` (or `nul` on Windows). On Unix-like systems, by default, the server's standard output and standard error are sent to pg_ctl's standard output (not standard error). The standard output of pg_ctl should then be redirected to a file or piped to another process such as a log rotating program like rotatelogs; otherwise `postgres` will write its output to the controlling terminal (from the background) and will not leave the shell's process group. On Windows, by default the server's standard output and standard error are sent to the terminal. These default behaviors can be changed by using `-l` to append the server's output to a log file. Use of either `-l` or output redirection is recommended.

In `stop` mode, the server that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the `-m` option. "Smart" mode (the default) waits for all active clients to disconnect and any online backup to finish. If the server is in hot standby, recovery and streaming replication will be terminated once all clients have disconnected. "Fast" mode does not wait for clients to disconnect and will terminate an online backup in progress. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all server processes immediately, without a clean shutdown. This will lead to a crash-recovery run on the next restart.

`restart` mode effectively executes a stop followed by a start. This allows changing the `postgres` command-line options. `restart` might fail if relative paths specified were specified on the command-line during server start.

`reload` mode simply sends the `postgres` process a SIGHUP signal, causing it to reread its configuration files (`postgresql.conf` , `pg_hba.conf` , etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

`status` mode checks whether a server is running in the specified data directory. If it is, the PID and the command line options that were used to invoke it are displayed. If the server is not running, the process returns an exit status of 3.

In `promote` mode, the standby server that is running in the specified data directory is commanded to exit recovery and begin read-write operations.

`kill` mode allows you to send a signal to a specified process. This is particularly valuable for Microsoft Windows which does not have a kill command. Use `--help` to see a list of supported signal names.

`register` mode allows you to register a system service on Microsoft Windows. The `-S` option allows selection of service start type, either "auto" (start service automatically on system startup) or "demand" (start service on demand).

`unregister` mode allows you to unregister a system service on Microsoft Windows. This undoes the effects of the `register` command.

Options

`-c``--core-file`

Attempt to allow server crashes to produce core files, on platforms where this is possible, by lifting any soft resource limit placed on core files. This is useful in debugging or diagnosing problems by allowing a stack trace to be obtained from a failed server process.

`-D` `_datadir_` `--pgdata` `_datadir_`

Specifies the file system location of the database configuration files. If this is omitted, the environment variable `PGDATA` is used.

`-l` `_filename_` `--log` `_filename_`

Append the server log output to `_filename_`. If the file does not exist, it is created. The umask is set to 077, so access to the log file is disallowed to other users by default.

`-m` `_mode_` `--mode` `_mode_`

Specifies the shutdown mode. `_mode_` can be `smart`, `fast`, or `immediate`, or the first letter of one of these three. If this is omitted, `smart` is used.

`-o` `_options_`

Specifies options to be passed directly to the `postgres` command.

The options should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

`-o` `_initdb-options_`

Specifies options to be passed directly to the `initdb` command.

The options should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

`-p` `_path_`

Specifies the location of the `postgres` executable. By default the `postgres` executable is taken from the same directory as `pg_ctl`, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the `postgres` executable was not found.

In `init` mode, this option analogously specifies the location of the `initdb` executable.

`-s` `--silent`

Print only errors, no informational messages.

`-t` `--timeout`

The maximum number of seconds to wait when waiting for startup or shutdown to complete. The default is 60 seconds.

`-V` `--version`

Print the pg_ctl version and exit.

`-w`

Wait for the startup or shutdown to complete. Waiting is the default option for shutdowns, but not startups. When waiting for startup, `pg_ctl` repeatedly attempts to connect to the server. When waiting for shutdown, `pg_ctl` waits for the server to remove its PID file. This option allows the entry of an SSL passphrase on startup. `pg_ctl` returns an exit code based on the success of the startup or shutdown.

`-W`

Do not wait for startup or shutdown to complete. This is the default for start and restart modes.

`-?` `--help`

Show help about pg_ctl command line arguments, and exit.

Options for Windows

`-N` `_servicename_`

Name of the system service to register. The name will be used as both the service name and the display name.

`-P` `_password_`

Password for the user to start the service.

`-S` `_start-type_`

Start type of the system service to register. start-type can be `auto`, or `demand`, or the first letter of one of these two. If this is omitted, `auto` is used.

`-U` `_username_`

User name for the user to start the service. For domain users, use the format `DOMAIN\username`.

Environment

`PGDATA`

Default data directory location.

`pg_ctl`, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14). For additional server variables, see postgres.

Files

`postmaster.pid`

The existence of this file in the data directory is used to help pg_ctl determine if the server is currently running.

`postmaster.opts`

If this file exists in the data directory, pg_ctl (in `restart` mode) will pass the contents of the file as options to postgres, unless overridden by the `-o` option. The contents of this file are also displayed in `status` mode.

Examples

Starting the Server

To start the server:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl start</kbd>
```

To start the server, waiting until the server is accepting connections:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl -w start</kbd>
```

To start the server using port 5433, and running without `fsync`, use:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl -o "-F -p 5433" start</kbd>
```

Stopping the Server

To stop the server, use:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl stop</kbd>
```

The `-m` option allows control over *how* the server shuts down:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl stop -m fast</kbd>
```

Restarting the Server

Restarting the server is almost equivalent to stopping the server and starting it again, except that `pg_ctl` saves and reuses the command line options that were passed to the previously running instance. To restart the server in the simplest form, use:

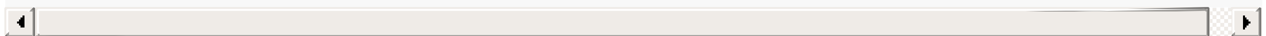
```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl restart</kbd>
```

To restart the server, waiting for it to shut down and restart:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl -w restart</kbd>
```

To restart using port 5433, disabling `fsync` upon restart:

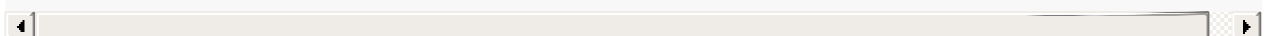
```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl -o "-F -p 5433" restart</kbd>
```



Showing the Server Status

Here is sample status output from pg_ctl:

```
<samp class="literal">$</samp> <kbd class="literal">pg_ctl status</kbd>
<samp class="literal">pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"</samp>
```



This is the command line that would be invoked in restart mode.

See Also

initdb, postgres

pg_resetxlog

Name

pg_resetxlog -- 重置一个数据库集群的预写日志以及其它控制内容

Synopsis

`pg_resetxlog` [`-f`] [`-n`] [`-o` `_oid_`] [`-x` `_xid_`] [`-e` `_xid_epoch_`] [`-m` `_mxid_` , `_mxid_`] [`-O` `_mxoff_`] [`-l` `_xlogfile_`] `_datadir_`

描述

`pg_resetxlog` 清理预写日志(WAL)并且可以有选择地重置其它一些存储在 `pg_control` 文件中的控制信息。有时候，如果这些文件崩溃了，就需要这个功能。 一定只把它用作最后的方法，就是说只有因为这样的崩溃导致服务器无法启动的时候才使用。

运行这个命令之后，可能就可以启动服务器了，但是， 一定要记住数据库可能因为部分提交的事务而含有不完整的数据。你应该马上转储数据， 运行 `initdb` ，然后重新加载。在重新加载之后， 检查不完整的部分然后根据需要进行修复。

这个命令只能由安装服务器的用户运行，因为它需要对数据目录的读写权限。 出于安全考虑， `pg_resetxlog` 不使用环境变量 `PGDATA` ， 你必须在命令行上声明数据目录。

如果 `pg_resetxlog` 抱怨说它无法判断用于 `pg_control` 的有效数据，那么你可以强制它继续处理，方法是声明 `-f` (强制)开关。 在这种情况下，那些丢失了的数据将用模糊的近似数值代替。大多数字段都可以匹配上， 但是下一个 OID 、下一个事务 ID 、下一个事务 ID 的 epoch(时间点)、 下一个多事务 ID(两阶段提交的东西)、下一个多事务偏移量、WAL 开始地址可能需要手工帮助， 这些字段可以使用下面讨论的选项设置。如果你不能判断所有这些字段的正确数值， 那么 `-f` 仍然可以使用，但是这样恢复过来的数据库正确性更值得怀疑： 立即转储和重新加载是必须的。在转储之前不要执行任何修改数据的操作， 因为任何这样的动作都可能把事情搞得更糟糕。

`-o` , `-x` , `-e` , `-m` , `-O` , `-l` 开关允许手工设置下一个 OID 、下一个事务 ID 、下一个事务 ID epoch 、 下一个和最旧的多事务 ID 、下一个多事务偏移量、WAL 起始位置的数值。 只有在 `pg_resetxlog` 无法通过读取 `pg_control` 判断合适的数值的时候才需要它。安全的数值可以用下面的方法判断：

- 对于下一个事务 ID(`-x`)而言，一个安全的数值是看看数据目录里的 `pg_clog` 里数值最大的文件名，然后加一，然后再乘上 1048576 。 请注意那些文件名是十六进制的。通常也以十六进制的形式声明选项值是最简单的。 比如，如果 `0011` 是 `pg_clog` 里最大的记录， `-x 0x1200000` 就可以了(后面的五个零提供了合适的乘积)。

- 下一个多事务 ID(`-m` 的第一部分)的安全值可以通过查看数据目录里 `pg_multixact/offsets` 子目录里面的数字最大的文件名，加一， 然后乘以 65536 得到。相反的，最老多事务ID（ `-m` 的第二部分） 的安全值可以通过查看相同目录里的数字最小的文件名，然后乘以65536得到。 和上面一样，文件名是十六进制的，因此最简单的方法是给选项声明一个十六进制的开关值， 然后在结尾加四个零。

- 下一个多事务偏移量(`-O`)的安全值可以通过检查数据目录里 `pg_multixact/members` 子目录下的数字最大的文件名，加一， 然后乘以 65536 得到。和上面一样，文件名是十六进制的。 这里没有像上面一样添加零的简单方法。

- WAL 的起始位置(`-l`)应该比目前存在于数据目录 `pg_xlog` 里面的任何WAL段文件号都大。它的文件名也是十六进制的，并且有三部分。 第一部分是"时间线 ID"，通常应该保持相同。比如， 如果 `00000001000000320000004A` 是 `pg_xlog` 里最大的条目， 那么选择 `-l 00000001000000320000004B` 或更多。

  > **Note:** `pg_resetxlog` 本身查看 `pg_xlog` 里面的文件， 并选择一个缺省的超过最后一个现存文件号的 `-l` 设置。因此， 只有知道WAL段文件当前不在 `pg_xlog` 中时，才需要手动调整 `-l` ， 例如离线归档中的条目；或如果 `pg_xlog` 的内容完全丢失。

- 没有很容易的办法来判断比数据库中最大的 OID 大一号的下一个 OID ， 不过很走运的是获取正确的下一个 OID 并非非常关键的事情。

- 除了由 `pg_resetxlog` 设定的字段外， 事务 ID epoch 实际上并未存储在数据库里的任何地方。 所以只要是涉及到数据库自身的任何数值都有效。你可能需要调整这个值以确保诸如 Slony-I之类的备份系统能够正常工作。如果是这样的话， 应当从下游已复制的数据库中获取恰当的值。

`-n` (无操作)选项指示 `pg_resetxlog` 打印从 `pg_control` 重新构造的数值然后不修改任何值就退出。这主要是一个调试工具， 但是在 `pg_resetxlog` 真正处理前进行的合理性检查的时候可能会有用。

`-V` 和 `--version` 选项打印pg_resetxlog 的版本然后退出。选项 `-?` 和 `--help` 显示参数的支持信息然后退出。

注意

在服务器运行的时候一定不要运行这个命令。如果发现在数据文件目录里有锁文件， 那么 `pg_resetxlog` 将拒绝启动。如果服务器崩溃， 那么可能会剩下一个锁文件；如果这样，你可以删除该锁文件以便允许 `pg_resetxlog` 运行。但是在你这么做之前， 一定要确保没有任何

后端服务器进程仍在运行。

postgres

Name

postgres -- PostgreSQL 数据库服务器

Synopsis

`postgres` [`_option_` ...]

描述

`postgres` 是PostgreSQL数据库服务器。 客户端应用程序为了访问数据库，将通过网络或本地连接到一个运行中的 `postgres` 进程。然后该 `postgres` 实例将启动一个独立的服务器进程来处理这个连接。

一个 `postgres` 总是管理来自同一个数据库集群的数据。 一个数据库集群是一组在同一个文件系统位置("数据区")存放数据的数据库。 一个系统上可以同时运行多个 `postgres` 进程， 只要他们使用不同的数据区和不同的端口号(见下文)。 `postgres` 启动时需要知道数据区的位置，该位置必须通过 `-D` 选项或 `PGDATA` 环境变量指定；没有缺省值。通常， `-D` 或 `PGDATA` 都直接指向由initdb创建的数据区。其他可能的文件布局在 Section 18.2里面有讨论。

缺省时 `postgres` 在前台启动并将日志信息输出到标准错误。 但在实际应用中， `postgres` 应当作为后台进程启动，而且多数是在系统启动时自动启动。

`postgres` 还能以单用户模式运行。这种用法主要用于initdb 的初始化过程中。有时候它也被用于调试或灾难性恢复。注意， 单用户模式运行的服务器并不适合于调试，因为没有实际的进程间通讯和锁动作发生。 当从 shell 上以单用户模式调用时，用户可以输入查询， 然后结果会在屏幕上以一种更适合开发者阅读(不适合普通用户)的格式显示出来。 在单用户模式下，将把会话用户 ID 设为 1 并赋予超级用户权限。该用户不必实际存在， 因此单用户模式运行的服务器可以用于对某些意外损坏的系统表进行手工恢复。

选项

`postgres` 接受下列命令行参数。关于这些选项的更详细讨论请参考 Chapter 18。你也可以通过设置一个配置文件来减少敲击这些选项。 有些(安全的)选项还可以从连接过来的客户端设置，以一种应用相关的方法仅对该会话生效。 比如，如果设置了 `PGOPTIONS` 环境变量，那么

基于libpq 的客户端就都把那个字符串传递给服务器，并被服务器解释成 `postgres` 命令行选项。

通用用途

`-A 0|1`

打开运行时断言检查，是检测编程错误的调试帮助。只有在编译PostgreSQL 时打开了它，你才能使用它。缺省是打开。

`-B` `_nbuffers_`

为服务器进程分配和管理的共享内存缓冲区数量。这个参数的缺省值是 initdb自动选择的；声明这个选项等同于设置 shared_buffers配置参数。

`-c` `_name_` = `_value_`

设置一个命名的运行时参数。PostgreSQL支持的配置参数在 Chapter 18里描述。大多数其它命令行选项实际上都是这样的参数赋值的短形式。 `-c` 可以出现多次从而设置多个参数。

`-C` `_name_`

打印命名的运行时参数的值，然后退出。（参阅上面的 `-c` 选项获取详细信息。） 这个选项可以在一个运行的服务器上使用，返回来自 `postgresql.conf` 的值， 通过在这个调用中提供的参数修改。当集群启动时，它不反应提供的参数。

这个选项本应为与服务器进程交互的其他程序，例如pg_ctl， 查询配置参数值。面向用户的应用应该使用xref linkend="sql-show">或 `pg_settings` 视图。

`-d` `_debug-level_`

设置调试级别。数值越高，写到服务器日志的调试输出越多。取值范围是 1 到 5 。 还可以针对某次单独的会话使用 `-d 0` 来防止从父 `postgres` 进程继承日志级别。

`-D` `_datadir_`

声明数据库配置文件的文件系统路径。细节详见Section 18.2。

`-e`

把缺省日期风格设置为"European"，也就是说用 `DMY` 规则解释日期输入， 并且在一些日期输出格式里日子在月份前面打印。参阅Section 8.5获取更多细节。

`-F`

关闭 `fsync` 调用以提高性能，但是要冒系统崩溃时数据毁坏的风险。 声明这个选项等效关闭了fsync参数。在使用之前阅读详细文档！

`-h` `_hostname_`

指定 `postgres` 侦听来自前端应用 TCP/IP 连接的 IP 主机名或地址。 数值也可以是一个用逗号分隔的地址列表，或者 `*` 表示监听所有可用的地址。 空值表示不监听任何 IP 地址，而只使用 Unix 域套接字与服务器连接。 缺省只监听localhost。 声明这个选项等效于设置listen_addresses配置参数。

`-i`

这个选项允许远程客户通过 TCP/IP(网际域套接字)与服务器通讯。没有这个选项， 服务器将只接受本地连接。这个选项等效于在 `postgresql.conf` 中或者通过 `-h` 选项将 `listen_addresses` 设为 `*` 。

这个选项已经废弃了，因为它不能实现listen_addresses的所有功能。 所以最好直接设置 `listen_addresses` 。

`-k` `_directory_`

指定 `postgres` 侦听来自前端应用连接的 Unix 域套接字的目录。 该值也可以是逗号分隔的目录列表。空值表明不监听任何 Unix 域套接字，在这种情况下， 只有TCP/IP套接字可以用来连接到服务器。缺省通常是 `/tmp` ， 但是可以在编译的时候修改。声明这个选项等同于设置unix_socket_directories 配置参数。

`-l`

这个选项使用SSL进行的安全通讯。要使用这个选项， 编译PostgreSQL时你必须打开了SSL支持。 有关使用SSL的信息，请参考Section 17.9。

`-N` `_max-connections_`

设置最多允许同时连接多少个客户端(也就是最多同时运行多少个服务器进程)。 这个参数的缺省值自动通过initdb选择。 声明这个选项等效于声明max_connections配置参数。

`-o` `_extra-options_`

在 `_extra-options_` 里面指定的命令行选项将被传递给所有由这个 `postgres` 派生的服务进程。 如果选项字符串包含任何空白，那么整个字符串必须用引号界定。

反对使用该选项，所有服务器进程的命令行选项都可以直接在 `postgres` 命令行上指定。

`-p` `_port_`

指定 `postgres` 侦听客户端连接的 TCP/IP 端口或本地 Unix 域套接字文件的扩展。 缺省的端口号是环境变量 `PGPORT` 的值。如果 `PGPORT` 没有设置， 那么缺省是 PostgreSQL 编译时指定的值(通常是 5432)。如果你声明了一个非缺省端口， 那么所有前端应用都必须用命令行选项或者 `PGPORT` 声明同一个端口。

`-s`

在每条命令结束时打印时间信息和其它统计信息。这个选项对测试性能和调节缓冲区数量有好处。

`-S` `_work-mem_`

声明内部排序和散列在求助于临时磁盘文件之前可以使用的内存数量。 参阅Section 18.4.1里描述的配置变量 `work_mem` 。

`-V``--version`

打印postgres版本然后退出。

`--``_name_` = `_value_`

设置一个命名的运行时参数；其缩写形式是 `-c` 。

`--describe-config`

以制表符分隔的 `COPY` 格式，导出服务器内部配置变量、描述、缺省值。 设计它主要是给管理工具使用。

`-?` `--help`

显示关于postgres命令行参数的帮助然后退出。

部分内部选项

这里描述的选项主要用于调试用途，并且在某些情况下帮助严重损坏的数据库恢复。 不应该在生产数据库的设置中使用这些选项。在这里列出只是给PostgreSQL 系统开发人员使用的。另外这些选项都可能在未来版本中改变或删除而不加说明。

`-f` `{ s | i | o | b | t | n | m | h }`

禁止某种扫描和连接方法的使用： `s` 和 `i` 分别关闭顺序和索引扫描， `o` , `b` 和 `t` 分别关闭只有索引扫描、位图索引扫描和TID扫描， 而 `n` , `m` , 和 `h` 分别关闭嵌套循环，融合(merge)和 Hash 连接。

顺序扫描和嵌套循环都不可能完全被关闭。 `-fs` 和 `-fn` 选项仅仅是在存在其它方法时阻碍优化器使用这些方法罢了。

`-n`

该选项主要用于调试导致服务器进程异常崩溃的问题。 对付这种情况的一般策略是通知所有其它服务器进程终止并重新初始化共享内存和信号灯。 这是因为一个出错的服务器进程可能在终止之前就已经对共享的东西造成了破坏。 该选项指定 `postgres` 不重新初始化共享数据结构。 一个有经验的系统程序员这时就可以使用调试器检查共享内存和信号灯状态。

`-O`

允许修改系统表的结构。这个参数用于 `initdb` 。

`-P`

读取系统表时忽略系统索引，但在更改数据时仍然更新索引。 这对于从索引已经损坏的系统表中恢复是很有帮助的。

`-t` `pa[rser] | pl[anner] | e[xecutor]`

打印与每个主要系统模块相关的查询记时统计。它不能和 `-s` 选项一起使用。

`-T`

该选项主要用于调试导致服务器进程异常崩溃的问题。 对付这种情况的一般策略是通知所有其它服务器进程终止并重新初始化共享内存和信号灯。 这是因为一个出错的服务器进程可能在终止之前就已经对共享的东西造成了破坏。 该选项指定 `postgres` 通过发送 `SIGSTOP` 信号停止其他所有服务器进程，但是并不让它们退出。 这样就允许系统程序员手动从所有服务器进程搜集内核转储。

`-v` `_protocol_`

声明这次会话使用的前/后服务器协议的版本数。该选项仅在内部使用。

`-W` `_seconds_`

一旦看见这个选项，进程就睡眠标出的秒数。这样就给开发者一些时间把调试器附着在该服务器进程上。

单用户模式的选项

下面的选项仅在单用户模式下可用。

`--single`

选中单用户模式。这个必须是命令行中的第一个选项。

`_database_`

要访问的数据库名字。这个必须是命令行中的最后一个选项。如果忽略掉则缺省为用户名。

`-E`

回显所有命令。

`-j`

禁止使用新行作为语句分隔符。

`-r` `_filename_`

将所有服务器输出日志保存到 `_filename_` 中。 在多用户模式下该选项将被忽略，所有进程都将使用stderr。

环境变量

`PGCLIENTENCODING`

客户端使用的缺省字符编码。客户端可以独立地覆盖它。这个值也可以在配置文件里设置。

`PGDATA`

缺省数据目录位置。

`PGDATESTYLE`

运行时参数DateStyle的缺省值。现在反对使用该环境变量。

`PGPORT`

缺省端口号(最好在配置文件中设置)。

`TZ`

服务器的时区。

诊断

一个提到了 `semget` 或 `shmget` 的错误信息可能意味着你需要重新配置你的内核， 提供足够的共享内存和信号灯。更多讨论，参阅Section 17.4。 你也可以通过降低shared_buffers值以减少PostgreSQL 的共享内存的消耗，或者降低max_connections值减少 PostgreSQL的信号灯的消耗。

如果碰到一个说另外一个服务器正在运行的错误信息，可以根据不同的系统使用命令

```
<samp class="literal">$</samp> <kbd class="literal">ps ax | grep postgres</kbd>
```

或

```
<samp class="literal">$</samp> <kbd class="literal">ps -ef | grep postgres</kbd>
```

如果确信没有冲突的服务器正在运行，那么你可以删除消息里提到的锁文件然后再次运行。

抱怨无法绑定端口的错误信息可能表明该端口已经被其它非PostgreSQL 进程使用。如果终止 `postgres` 后又马上用同一个端口运行它， 也可能得到这个错误信息；这时，你必须多等几秒，等操作系统关闭了该端口后再试。 最后，如果你使用了一个操作系统认为是保留的端口，也可能导致这个错误信息。例如， 许多 Unix 版本认为低于 1024 的端口号是"可信任的"， 因而只有 Unix 超级用户可以使用它们。

注意

pg_ctl工具可以用于安全而有效地启停 `postgres` 服务器。

如果有可能，不要使用 `SIGKILL` 杀死主 `postgres` 服务器进程。这样会阻止 `postgres` 在退出前释放它持有的系统资源(例如共享内存和信号灯)。这样可能会影响到将来启动新的 `postgres` 进程。

可以使用 `SIGTERM`，`SIGINT`，`SIGQUIT` 信号正常结束 `postgres` 服务器进程。 第一个信号将等待所有的客户端退出后才退出。第二个将强制断开所有客户端， 而第三个将不停止立刻退出， 导致在重启时的恢复运行。

`SIGHUP` 会重新加载服务器配置文件。 也可以向一个单独的服务器进程发送 `SIGHUP` 信号，但是这样做没什么意义。

要取消一个正在运行的查询，可以向正在执行该查询的进程发送 `SIGINT` 信号。 要终止一个后端进程，可以向那个进程发送 `SIGTERM` 信号。也可以参阅 Section 9.26.2里面的 `pg_cancel_backend` 和 `pg_terminate_backend` 获取这两个动作的相等SQL调用。

`postgres` 服务器发送 `SIGQUIT` 信号告诉子服务器进程立即退出且不做清理工作，用户应当尽量避免 使用该信号。同时， 发送 `SIGKILL` 信号也是不明智的： 主 `postgres` 进程将把这个信号当作崩溃信号， 然后会强制其他兄弟进程作为标准的崩溃回复过程退出。

臭虫

`--` 选项在FreeBSD或 OpenBSD上无法运行，应该使用 `-c` 。 这在受影响的系统里是个臭虫；如果这个毛病没有修补好， 将来的PostgreSQL版本将提供一个绕开的办法。

用法

启动一个单用户模式的服务器：

```
<kbd class="literal">postgres --single -D /usr/local/pgsql/data `_other-options_` my_data
```

用 `-D` 给服务器提供正确的数据库目录的路径， 或者确保环境变量 `PGDATA` 已经正确设置。同时还要声名你想用的特定数据库名字。

通常， 单用户模式的服务器把换行符当做命令输入完成字符；它还不懂分号的作用， 因为那些东西是在psql里的。要想把一行分成多行写， 你必需在除最后一个换行符以外的每个换行符前面敲一个反斜杠。

但是如果使用了 `-j` 命令行选项，新行将不被当作命令结束符。 此时服务器将从标准输入一直读取到EOF标志为止， 然后把所有读到的内容当作一个完整的命令字符串看待， 并且反斜杠与换行符也被当作普通字符来看待。

输入EOF(**Control+D**) 即可退出会话。如果你已经使用了 `-j` 则必须连续使用两个EOF才行。

请注意单用户模式运行的服务器不会提供复杂的行编辑功能(比如，没有命令行历史)。 单用户模式也不做任何后台处理，像自动检查点。

例子

用缺省值在后台启动 `postgres` ：

```
<samp class="literal">$</samp> <kbd class="literal">nohup postgres >logfile 2>&1 </dev/nu
```

在指定的端口启动 `postgres` ， 如1234：

```
<samp class="literal">$</samp> <kbd class="literal">postgres -p 1234</kbd>
```

要连接到这个服务器使用psql，用 -P 选项指定这个端口：

```
<samp class="literal">$</samp> <kbd class="literal">psql -p 1234</kbd>
```

或者设置环境变量 `PGPORT` ：

```
<samp class="literal">$</samp> <kbd class="literal">export PGPORT=1234</kbd>
<samp class="literal">$</samp> <kbd class="literal">psql</kbd>
```

命名的运行时参数可以用下列的风格之一设置：

```
<samp class="literal">$</samp> <kbd class="literal">postgres -c work_mem=1234</kbd>
<samp class="literal">$</samp> <kbd class="literal">postgres --work-mem=1234</kbd>
```

两种形式都覆盖那些现有的在 `postgresql.conf` 里面的 `work_mem` 设置。 请注意在参数名里的下划线在命令行上可以写成下划线，也可以写成连字符。 除了用于短期的实验以外，更好的习惯是编辑 `postgresql.conf` 里面的设置， 而不是倚赖命令行开关设置参数。

又见

initdb, pg_ctl

postmaster

Name

postmaster -- PostgreSQL 数据库服务器

Synopsis

`postmaster` [`_option_` ...]

描述

`postmaster` 是 `postgres` 的别名，反对使用。

又见

postgres

VII. 内部

这部分包含可以用于 PostgreSQL 开发人员的分类信息。

Table of Contents

Chapter 46. PostgreSQL内部概述

Table of Contents

> 作者: 本章最初是*Enhancement of the ANSI SQL Implementation of PostgreSQL*的一部分，它是 Stefan Simkovics 在维也纳理工大学写的硕士论文， 是由 O.Univ.Prof.Dr. Georg Gottlob 和 Univ.Ass. Mag. Katrin Seyr 指导的。

本章给出了PostgreSQL后端服务器的内部结构的一个概貌。 在阅读完毕下面的章节后，你应该对查询是如何处理的有一个概念了。 本章并不准备提供对PostgreSQL内部操作的详细描述， 因为这样的一份文档将会非常庞大。 本章只是试图帮助读者了解从后端收到查询后到结果返回给客户端之间一般操作顺序。

46.1. 查询经过的路径

下面是一个简短的描述，描述一个查询从开始到得到结果要经过的阶段。

1. 首先必须先建立起从应用程序到PostgreSQL服务器的连接。 应用程序向服务器发送查询然后等待接收从服务器返回的结果。

2. 分析器阶段检查从应用程序(客户端)发送过来的查询， 核对语法并创建一个查询树。

3. 重写系统接收分析阶段来的查询树且搜索任何应用到查询树上的 规则(存储在系统表里)， 并根据给出的 规则体进行转换。

   重写系统的一个应用就是实现视图。 当一个查询访问一个视图时(也就是一个虚拟表)，重写系统改写用户的查询， 使之成为一个访问在视图定义里给出的对基本表的查询。

4. 规划器/优化器接收(改写后的)查询树然后创建一个查询规划， 这个查询规划是执行器的输入。

   它(规划器/优化器)首先创建所有得出相同结果的可能的路径。 例如，如果待扫描的关系上有一个索引，那么扫描的路径就有两个。一个可能是简单的顺序查找， 而另一个可能就是使用索引的查找。下一步是计算出不同路径的执行开销， 并且选择和返回开销最少的那条。开销最小的路径然后会被展开成为一个可以供执行器使用的完整的查询规划。

5. 执行器递归地走过规划树并且按照规划指定的方式检索数据行。 执行器在对关系进行扫描时使用存储系统进行排序 和连接，计算条件并且最终交回生成的数据行。

在随后的小节里，将对上面的每一个步骤进行更详细的讨论，以便让对PostgreSQL 的内部控制和数据结构有一个更准确的理解。

46.2. 连接是如何建立起来的

PostgreSQL是用一个简单的"每用户一进程" 的 client/server 模型实现的。在这种模式里一个客户端进程 只与恰好一个服务器进程连接。因为不知道具体要建立多少个连接， 所以不得不利用一个主进程在每次连接请求时派生出一个新的服务器进程来。 这个主进程叫做 `postgres` ，它监听着一个特定的 TCP/IP 端口等待进来的连接。 每当检测到一个连接请求时， `postgres` 进程派生出一个新的服务器进程。 服务器进程之间使用信号灯和共享内存进行通讯， 以确保在并发的数据访问过程中的数据完整性。

客户端进程可以是任何理解PostgreSQL协议(在Chapter 48 里描述)的程序。许多客户端都是基于 C 语言库libpq的程序， 但是也存在几个对协议独立的实现，比如 Java JDBC驱动。

一旦建立起来连接，客户端进程就可以向后端(服务器)进程发送查询了。 查询是通过纯文本传输的，也就是说在前端(客户端)不做任何分析处理。服务器分析查询， 创建执行规划，执行该规划并且通过已经建立起来的连接把检索出来的数据行返回给客户端。

46.3. 分析器阶段

分析器阶段含两个部分：

- 在 `gram.y` 和 `scan.l` 里定义的分析器 是使用 Unix 工具bison和flex创建的。
- 转换处理对分析器返回的数据结构进行修改和增补。

46.3.1. 分析器

分析器必须检查(以纯 ASCII 文本方式到来的)查询字符串的语法。如果语法正确， 则创建一个分析树并将之传回，否则，返回一个错误。 实现分析器和词法器使用了著名的 Unix 工具bison和flex。

词法器在文件 `scan.l` 里定义， 负责识别标识符和 *SQL* 关键字等。 对于发现的每个关键字或者标识符都会生成一个记号并且传递给分析器。

分析器在文件 `gram.y` 里定义并且包含一套语法规则 和触发规则时执行的动作。动作代码(实际上是 C 代码)用于建立分析树。

文件 `scan.l` 用flex转换成 C 源文件 `scan.c` ，而 `gram.y` 用bison 转换成 `gram.c` 。在完成这些转换后，一个通用的 C 编译器就可以用于创建分析器。 千万不要对生成的 C 源文件做修改，因为下一次调用flex 或bison时会把它们覆盖。

> **Note:** 上面提到的转换和编译是使用跟随PostgreSQL发布的 *makefiles*自动完成的。

对bison或者 `gram.y` 里的语法规则的详细描述超出本文的范围。有很多关于flex 和bison的书籍和文档。你在开始研究 `gram.y` 里给出的语法之前应该对bison很熟悉，否则你是看不懂那里面的内容， 理解不了发生了什么事情的。

46.3.2. 转换处理

分析器阶段只使用和 SQL 语法结构相关的固定规则创建一个分析树。它不会查找任何系统表， 因此就不可能理解请求查询里面的详细的语意。在分析器技术之后， 转换处理接受分析器传过来的分析树然后做进一步处理，解析哪些查询中引用了哪个表、 哪个函数、哪个操作符的语意。所生成的表示这个信息的数据结构叫做查询树。

把裸分析和语意分析分成两个过程的原因是系统表查找只能在一个事务中进行， 而不想在一接收到查询字符串就发起一个事务。裸分析阶段已经足够可以标识事务控制命令(`BEGIN` ， `ROLLBACK` 等)，并且这些东西不用任何进一步的分析就可以执行。 一旦知道正在处

理一个真正的查询(比如 `SELECT` 或 `UPDATE`)， 就可以发起一个事务了(如果还没开始这么一个)。只有这个时候可以调用转换处理。

转换处理生成的查询树结构上在很大程度上类似于裸分析树，但是在细节上有很多区别。 比如，在分析树里的 `FuncCall` 节点代表那些看上去像函数调用的东西。 根据引用的名字是一个普通函数还是一个聚集函数，这个可能被转换成一个 `FuncExpr` 或 `Aggref` 节点。同样，有关字段和表达式结果的具体数据类型也添加到查询树中。

46.4. PostgreSQL规则系统

PostgreSQL有一个强大的规则系统， 用以描述视图和不明确的视图更新。 最初的PostgreSQL规则系统由两个实现组成：

- 第一个能用的规则系统采用行级别的处理， 是在执行器的深层实现的。每次访问一条独立的行时都要调用规则系统。 这个实现在 1995 年被删除了，那时伯克力 Postgres 项目的最后一个官方版本正转换成Postgres95。
- 第二个规则系统的实现从技术角度来说叫查询重写。 重写系统是一个存在于分析器阶段和 规划器/优化器之间的一个模块。这个技术实现仍然存在。

查询重写在Chapter 38里有比较详细的讨论，所以无需再次介绍。 只需要说明重写器的输入和输出都是查询树，也就是说，在树的语意细节的表现或者层次方面没有变化。 可以把重写系统当作某种宏展开的机制。

46.5. 规划器/优化器

规划器/优化器的任务是创建一个优化执行规划。 一个特定的 SQL 查询(因此也就是一个查询树)实际上可以以多种不同的方式执行， 每种都生成相同的结果集。如果可能， 查询优化器将检查每个可能的执行规划， 最终选择认为运行最快的执行计划。

> **Note:** 有些情况下，检查一个查询所有可能的执行方式会花去很多时间和内存空间。 特别是在正在执行的查询涉及大量连接操作的时候。 为了在合理的时间里判断一个合理的(而不是优化的)查询计划。PostgreSQL 当连接的数量超过一个阈值(参阅geqo_threshold)时使用 基因查询优化器(参阅Chapter 53)。

规划器的搜索过程实际上是与叫做*paths*的数据结构一起结合运转的， 这个数据结构是一个很简单的规划的精简版本，它只包括规划器用来决策所必须的信息。 在找到最经济的路径之后，就制作一个完整的规划树传递给执行器。 它有足够的详细信息，代表着需要执行的计划，执行器可以读懂并运行之。在本章剩余部分， 将忽略路径和规划之间的区别。

46.5.1. 生成可能的规划

规划器/优化器通过为扫描查询里出现的每个关系(表)生成规划。 可能的规划是由每个关系上有哪些可用的索引决定的。对一个关系总是可以进行一次顺序查找， 所以总是会创建只使用顺序查找的规划。假设一个关系上定义着一个索引(例如 B-tree 索引)， 并且一条查询包含约束 `relation.attribute OPR constant` 。 如果 `relation.attribute` 碰巧匹配 B-tree 索引的关键字并且 `OPR` 又是列出在索引的操作符类中的操作符中的一个， 那么将会创建另一个使用 B-tree 索引扫描该关系的规划。如果还有别的索引， 而且查询里面的约束又和那个索引的关键字匹配，则还会生成更多的规划。也会为索引生成索引扫描规划， 这个规划有一种可以匹配查询的 `ORDER BY` 子句（如果有）的顺序， 或者有一种可能对融合加入（见下面）有用的顺序。

如果查询需要加入两个或者更多的关系，在所有的对单一关系的扫描可行的规划被发现后，连接各个关系的规划就要被考虑了。有三种可能的连接策略：

- 嵌套循环连接：对左边的关系里面找到的每条行都对右边关系进行一次扫描。 这个策略容易实现，但是可能会很耗费时间。（但是，如果右边的关系可以用索引扫描， 那么这个可能就是个好策略。可以用来自左边关系的当前行的数值为键字进行对右边关系的索引扫描。）
- 融合连接：在连接开始之前，每个关系都对连接字段进行排序。 然后对两个关系并发扫描，匹配的行就组合起来形成连接行。这种联合更有吸引力， 因为每个关系都只用扫描一次。要求的排序步骤可以通过明确的排序步骤， 或者是使用连接键字上的索引按照恰当的顺序扫描关系。

- *Hash* 连接：首先扫描右边的关系， 并用连接的字段作为散列键字加载进入一个 Hash 表，然后扫描左边的关系， 并将找到的每行用作散列键字来以定位表里匹配的行。

如果查询里的关系多于两个，最后的结果必须通过一个连接步骤树建立， 每个步骤有两个输入。规划器检查不同的连接顺序可能，找出开销最小的。

如果查询使用少于geqo_threshold的关系， 一个近乎详尽的查询用来进行查找最好的连接顺序。规划器优先的考虑介于任何两个关系间的连接子句， 在 `WHERE` 条件中存在一个匹配的连接子句（例如，存在像 `where rel1.attr1=rel2.attr2` 这样的约束）。 没有连接子句的连接对只有在没有别的选择的时候才考虑，也就是说， 一个关系没有和任何其它关系的连接子句可用。所有可能的规划都是为每个被规划器考虑的链接对生成的， 并且那个（预计是）最经济的被选择。

当 `geqo_threshold` 溢出时，连接序列被认为是由启发式方法决定的， 描述在Chapter 53。否则，流程是相同的。

完成的查询树由对基础关系的顺序或者索引扫描组成，并根据需要加上嵌套循环、融合、以及 Hash 连接节点， 加上任何需要的辅助步骤，比如排序节点或者聚集函数计算节点等。 大多数这些规划节点类型都有额外的做选择 (抛弃那些不符合指定布尔条件的行)和投影 (基于给出的字段数值计算一个派生出的字段集， 也就是在需要时计算标量表达式)。规划器的一个责任是从 `WHERE` 子句中附加选择条件以及为规划树最合适的节点计算所需要的输出表达式。

46.6. 执行器

执行器接受规划器/优化器创建的查询规划然后递归地处理它， 抽取所需要的行集合。它实际上是一个需求-拉动地流水线机制。每次调用一个规划节点地时候， 它都必须给出更多的一个行，或者汇报它已经完成行的传递了。

为了提供一个具体的例子，假设顶端节点是一个 `MergeJoin` 节点。 在做任何融合之前，首先得抓取两行(每个子规划一行)。 因此执行器递归地调用自己来处理子规划(它从附着在 `lefttree` 上的子规划开始)。 新的顶端节点(左子规划的顶端节点)假设是，一个 `Sort` 节点， 然后还是需要递归地获取一个输入行。 `Sort` 节点的子节点可能是一个 `SeqScan` 节点，代表对一个表的实际读取动作。这个节点的执行导致执行器从表中抓取一行然后把它返回给调用的节点。 `Sort` 将不断调用它的子节点以获取需要排序的所有行。 在用尽输入之后(由子节点返回一个 NULL 而不是一行表示)， `Sort` 代码执行排序， 然后就可以返回它的第一个输出行，也就是按照排序顺序输出的第一行。它仍然保持剩下的行的排序状态， 这样在随后有需求的时候，它就可以按照排序顺序返回这些行。

`MergeJoin` 节点也会类似地要求从它的右边子规划获取第一行。 然后它比较这两行看看它们是否能连接；如果能，那么它给它的调用者返回一个连接行。 在下一次调用的时候，或者是在它无法连接当前的两行的时候就是这次调用的时候， 它抓取其中一个表的下一行(抓取哪个表取决于比较结果如何)，然后再检查看看两个表是否匹配。 最后，其中一个子规划耗尽资源，而 `MergeJoin` 返回 NULL， 表明无法继续生成更多的连接行。

复杂的查询可能包含许多层的规划节点，但是一般的过程都是一样的： 每个节点在每次被调用的时候都计算并返回它的下一个输出行。 每个节点同样负责附加上任何规划器赋予它的选择或者投影表达式。

执行器机制是用于计算所有的四种基本 SQL 查询类型的： `SELECT` ， `INSERT` ， `UPDATE` 和 `DELETE` 。对于 `SELECT` 而言， 顶层的执行器代码只是需要发送查询规划树返回的每一行给客户端。对于 `INSERT` ， 返回的每一行都插入到 `INSERT` 声明的目标表中。也就是在一个名为 `ModifyTable` 的特殊的顶级规划节点执行。（一个简单的 `INSERT ... VALUES` 命令创建一个简单的规划树， 包含一个 `Result` 节点，它只计算得出一个结果行，并且 `ModifyTable` 执行这个插入。 但是 `INSERT ... SELECT` 可能需要执行器的全部能力。）对于 `UPDATE` ， 规划器安排每个计算出来的行都包括所有更新的字段，加上原来的目标行的 *TID*(元组ID或行ID)； 这些数据输入到一个 `ModifyTable` 节点，这个节点使用这些信息创建一个新的更新过的行， 并且标记旧行被删除。对于 `DELETE` ，规划实际上返回的唯一的一个字段是 TID ， 然后 `ModifyTable` 节点简单地使用这个 TID 访问每个目标行，并且把它们标记为已删除。

Chapter 47. 系统表

Table of Contents

系统表是关系型数据库管理系统存放结构元数据的地方，比如表和字段以及内部登记信息等。 PostgreSQL的系统表就是普通表。你可以删除然后重建这些表、 增加列、插入和更新数值，然后彻底把系统搞垮。不应该手工修改系统表， 通常总有 SQL 命令可以做这些事情。比如， `CREATE DATABASE` 向 `pg_database` 表插入一行，并且实际上在磁盘上创建该数据库。有几种特别深奥的操作例外，比如增加索引访问方法。

47.1. 概述

Table 47-1列出了系统表。每个表更详细的文档在后面。

大多数系统表都是在数据库创建的过程中从模版数据库中拷贝过来的，因此都是数据库相关的。 少数表是在一个集群中物理上所有数据库共享的；这些表在独立的表的描述中用指明了。

Table 47-1. 系统表

表名字	用途
pg_aggregate	聚集函数
pg_am	索引访问方法
pg_amop	访问方法操作符
pg_amproc	访问方法支持过程
pg_attrdef	字段缺省值
pg_attribute	表的列("属性")
pg_authid	认证标识符(角色)
pg_auth_members	认证标识符成员关系
pg_cast	转换(数据类型转换)
pg_class	表、索引、序列、视图 ("关系")
pg_constraint	检查约束、唯一约束、主键约束、外键约束
pg_collation	排序规则 (本地信息)
pg_conversion	编码转换信息
pg_database	本集群内的数据库
pg_db_role_setting	每个角色和每个数据库设置
pg_default_acl	对象类型的缺省权限
pg_depend	数据库对象之间的依赖性
pg_description	数据库对象的描述或注释
pg_enum	定义枚举标签和值
pg_event_trigger	事件触发器
pg_extension	安装扩展
pg_foreign_data_wrapper	定义外部数据封装器

pg_foreign_server	定义外部服务器
pg_foreign_table	附加的外部表信息
pg_index	附加的索引信息
pg_inherits	表继承层次
pg_language	用于写函数的语言
pg_largeobject	大对象的数据页
pg_largeobject_metadata	大对象的元数据
pg_namespace	模式
pg_opclass	访问方法操作符类
pg_operator	操作符
pg_opfamily	访问方法操作符族
pg_pltemplate	过程语言使用的模板数据
pg_proc	函数和过程
pg_range	有关范围类型的信息
pg_rewrite	查询重写规则
pg_seclabel	数据库对象上的安全标签
pg_shdepend	在共享对象上的依赖性
pg_shdescription	共享对象上的注释
pg_shseclabel	共享数据对象上的安全标签
pg_statistic	优化器统计
pg_tablespace	这个数据库集群里面的表空间
pg_trigger	触发器
pg_ts_config	文本搜索配置
pg_ts_config_map	文本搜索配置的标记映射
pg_ts_dict	文本搜索字典
pg_ts_parser	文本搜索解析器
pg_ts_template	文本搜索模板
pg_type	数据类型
pg_user_mapping	对外部服务器进行映射的用户

47.2. pg_aggregate

存储与聚集函数有关的信息。聚集函数是对一个数值集(通常每个匹配查询条件的行中的一个字段) 进行操作的函数，它返回从这些值中计算出的一个数值。典型的聚集函数是 sum ，count ， max 。 pg_aggregate 里的每条记录都是一条 pg_proc 里面的记录的扩展。 pg_proc 记录承载该聚集的名字、输入和输出数据类型，以及其它一些和普通函数类似的信息。

Table 47-2. pg_aggregate 字段

名字	类型	引用	描述
aggfnoid	regproc	pg_proc .oid	此聚集函数的 pg_proc OID
aggtransfn	regproc	pg_proc .oid	转换函数
aggfinalfn	regproc	pg_proc .oid	最终处理函数(如果没有则为零)
aggsortop	oid	pg_operator .oid	关联排序操作符(如果没有则为零)
aggtranstype	oid	pg_type .oid	此聚集函数的内部转换(状态)数据的数据类型
agginitval	text	转换状态的初始值。这是一个文本数据域，它包含初始值的外部字符串表现形式。 如果数据域是 null ， 那么转换状态值从 null 开始。	

新聚集函数是用CREATE AGGREGATE命令注册的。参阅Section 35.10 获取关于编写聚集函数以及转换函数的含义等的更多信息。

47.3. pg_am

pg_am 存储有关索引访问方法的信息。系统支持的每种索引访问方法都有一行。 这个表的内容在Chapter 54详细讨论。

Table 47-3. pg_am 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
amname	name	访问方法的名字	
amstrategies	int2	这个访问方法的操作符策略个数，或者如果访问方法没有一个固定的操作符策略集则为0。	
amsupport	int2	这个访问方法的支持过程个数	
amcanorder	bool	这种访问方式是否支持通过索引字段值的命令扫描排序？	
amcanorderbyop	bool	这种访问方式是否支持通过索引字段上操作符的结果的命令扫描排序？	
amcanbackward	bool	这种访问方式是否支持向后扫描？	
amcanunique	bool	这种访问方式是否支持唯一索引?	
amcanmulticol	bool	这种访问方式是否支持多字段索引?	
amoptionalkey	bool	这种访问方式是否支持第一个索引字段上没有任何约束的扫描?	
amsearcharray	bool	这种访问方式是否支持 ScalarArrayOpExpr 搜索?	
amsearchnulls	bool	这种访问方式是否支持 IS NULL / NOT NULL 搜索?	
amstorage	bool	允许索引存储的数据类型与列的数据类型不同?	
amclusterable	bool	允许在一个这种类型的索引上群集?	
ampredlocks	bool	允许这种类型的一个索引管理	

amkeytype	oid	pg_type .oid	存储在索引里的数据的类型，如果不是一个固定的类型则为0
aminsert	regproc	pg_proc .oid	"插入这个行" 函数
ambeginscan	regproc	pg_proc .oid	"准备索引扫描" 函数
amgettuple	regproc	pg_proc .oid	"下一个有效行" 函数, 如果没有则为0
amgetbitmap	regproc	pg_proc .oid	"抓取所有有效行" 函数, 如果没有则为0
amrescan	regproc	pg_proc .oid	"（重新）开始索引扫描" 函数
amendscan	regproc	pg_proc .oid	"索引扫描后清理" 函数
ammarkpos	regproc	pg_proc .oid	"标记当前扫描位置" 函数
amrestrpos	regproc	pg_proc .oid	"恢复已标记的扫描位置" 函数
ambuild	regproc	pg_proc .oid	"建立新索引" 函数
ambuildempty	regproc	pg_proc .oid	"建立空索引" 函数
ambulkdelete	regproc	pg_proc .oid	批量删除函数
amvacuumcleanup	regproc	pg_proc .oid	VACUUM 后的清理函数
amcanreturn	regproc	pg_proc .oid	检查是否索引支持唯一索引扫描的函数，如果没有则为0
amcostestimate	regproc	pg_proc .oid	估计一个索引扫描开销的函数
amoptions	regproc	pg_proc .oid	为一个索引分析和确认 reloptions 的函数

47.4. pg_amop

`pg_amop` 表存储有关和访问方法操作符族关联的信息。 如果一个操作符是一个操作符族中的成员，那么在这个表中会占据一行。 一个族成员是一个*search*操作符或一个*ordering*操作符。 一个操作符可以在多个族中出现，但是不能在一个族中的多个搜索位置或多个排序位置中出现。 （尽管不太可能，但这是允许的，一个操作符可以被搜索和排序目的使用。）

Table 47-4. `pg_amop` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性；必须明确选择)	
`amopfamily`	`oid`	`pg_opfamily` .oid	这个项的操作符族
`amoplefttype`	`oid`	`pg_type` .oid	操作符的左输入类型
`amoprighttype`	`oid`	`pg_type` .oid	操作符的右输入类型
`amopstrategy`	`int2`	操作符策略数	
`amoppurpose`	`char`	操作符目的, `s` 为搜索或 `o` 为排序	
`amopopr`	`oid`	`pg_operator` .oid	该操作符的OID
`amopmethod`	`oid`	`pg_am` .oid	索引访问方式操作符族
`amopsortfamily`	`oid`	`pg_opfamily` .oid	如果是一个排序操作符，则为这个项排序所依据的btree操作符族； 如果是一个搜索操作符，则为0

"搜索"操作符表明这个操作符族的一个索引可以被搜索，找到所有满足 `WHERE` `_indexed_column_` `_operator_` `_constant_` 的行。 显然，这样的操作符必须返回布尔值，并且它的左输入类型必须匹配索引的字段数据类型。

"排序"操作符表明这个操作符族的一个索引可以被扫描，返回以 `ORDER BY` `_indexed_column_` `_operator_` `_constant_` 顺序表示的行。这样的操作符可以返回任意可排序的数据类型，它的左输入类型也必须匹配索引的字段数据类型。 `ORDER BY` 的确切的语义是由 `amopsortfamily` 字段指定的， 该字段必须为操作符的返回类型引用一个btree操作符族。

> **Note:** 目前，假设排序操作符的排序顺序是被族缺省引用的，也就是是 `ASC NULLS LAST` 。 这可以通过添加附加的行来明确声明排序选项来释放。

一个项的 `amopmethod` 必须匹配它包含的操作符族的 `opfmethod` （包括 `amopmethod` 是故意违反性能原因的表结构的规范化）。同样， `amoplefttype` 和 `amoprighttype` 必须匹配引用的 `pg_operator` 的 `oprleft` 和 `oprright` 。

47.5. `pg_amproc`

`pg_amproc` 存储有关与访问方法操作符族相关联的支持过程的信息。 每个属于某个操作符族的支持过程都占有一行。

Table 47-5. `pg_amproc` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`amprocfamily`	`oid`	`pg_opfamily` .oid	该项的操作符族
`amproclefttype`	`oid`	`pg_type` .oid	相关操作符的左输入数据类型
`amprocrighttype`	`oid`	`pg_type` .oid	相关操作符的右输入数据类型
`amprocnum`	`int2`	支持过程编号	
`amproc`	`regproc`	`pg_proc` .oid	过程的 OID

`amproclefttype` 和 `amprocrighttype` 字段的习惯解释， 他们标识一个特定支持过程支持的操作符的左和右输入类型。对于某些访问方式， 他们匹配支持过程本身的输入数据类型，对其他的则不这样。有一个对索引的"缺省" 支持过程的概念， `amproclefttype` 和 `amprocrighttype` 都等于索引操作符类的 `opcintype` 。

47.6. `pg_attrdef`

`pg_attrdef` 表存储字段缺省值。字段的主要信息存放在 `pg_attribute` (见下文)。只有明确声明一个缺省值(该表何时创建或字段何时增加)的字段在这里有行。

Table 47-6. `pg_attrdef` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`adrelid`	`oid`	`pg_class` .oid	这个字段所属的表
`adnum`	`int2`	`pg_attribute`.attnum	字段数目
`adbin`	`pg_node_tree`	字段缺省值的内部表现形式	
`adsrc`	`text`	人类可读的缺省值的内部表现形式	

`adsrc`是历史遗留，最好不要使用它， 因为它并未跟踪可能影响缺省值表现形式的外部变化。反编译`adbin` 字段(比如，用`pg_get_expr`)是更好的显示缺省值的方法。

47.7. `pg_attribute`

`pg_attribute` 表存储关于表的字段的信息。数据库里每个表的每个字段都在 `pg_attribute` 里有一行。还有用于索引，以及所有在 `pg_class` 里有记录的对象。

术语属性等效于列/字段，使用它是历史原因。

Table 47-7. `pg_attribute` 字段

名字	类型	引用	描述
`attrelid`	`oid`	`pg_class`.oid	此字段所属的表
`attname`	`name`	字段名字	
`atttypid`	`oid`	`pg_type`.oid	这个字段的数据类型
`attstattarget`	`int4`	控制ANALYZE为这个字段积累的统计细节的级别。零值表示不收集统计信息。负数表示使用系统缺省的统计对象。正数值的确切信息是和数据类型相关的。对于标量数据类型， `attstattarget` 既是要收集的"最常用数值" 的目标数目，也是要创建的柱状图的目标数量。	
`attlen`	`int2`	是本字段类型的 `pg_type.typlen` 的拷贝	
`attnum`	`int2`	字段数目。普通字段是从 1 开始计数的。系统字段 (比如 `oid`)有(任意)负数。	
`attndims`	`int4`	如果该字段是数组，那么是维数，否则是 0 。目前，一个数组的维数并未强制， 因此任何非零值都表示"这是一个数组"。	
`attcacheoff`	`int4`	在磁盘上的时候总是 -1 ， 但是如果加载入内存中的行描述器中， 它可能会被更新以缓冲在行中字段的偏移量。	
`atttypmod`	`int4`	记录创建新表时支持的类型特定的数据(比如一个 `varchar` 字段的最大长度)。 它传递给类型相关的输入函数和长度转换函数当做第三个参数。 其值对那些不需要 `atttypmod` 的类型通常为 -1 。	
`attbyval`	`bool`	这个字段类型的 `pg_type.typbyval` 的拷贝。	
		这个字段的类型的 `pg_type.typstorage` 的	

attstorage	char	这个字段可以在字段创建之后改变，以便于控制存储策略。	
attalign	char	这个字段类型的 pg_type.typalign 的拷贝	
attnotnull	bool	这代表一个非空约束。可以改变这个字段以打开或者关闭这个约束。	
atthasdef	bool	这个字段有一个缺省值，此时它对应 pg_attrdef 表里实际定义此值的记录。	
attisdropped	bool	这个字段已经被删除了，不再有效。一个已经删除的字段物理上仍然存在表中， 但会被分析器忽略，因此不能再通过 SQL 访问。	
attislocal	bool	这个字段是局部定义在关系中的。请注意一个字段可以同时是局部定义和继承的。	
attinhcount	int4	这个字段所拥有的直接祖先的个数。如果一个字段的祖先个数非零， 那么它就不能被删除或重命名。	
attcollation	oid	pg_collation .oid	这个字段定义的排序规则，如果这个字段不是排序规则数据类型则为0。
attacl	aclitem[]	如果在这个字段上明确的获得任意，则为字段级访问权限。	
attoptions	text[]	属性级选项，使用"keyword=value"格式的字符串	
attfdwoptions	text[]	属性级外部数据封装器选项，使用"keyword=value"格式的字符串	

在一个已被删除字段的 pg_attribute 记录里， atttypid 将被重置为零，但是 attlen 和其它从 pg_type 拷贝的仍然有效。 这么安排是为了对付后来被删除的字段的数据类型也被删除的情况，因为这个时候不再有 pg_type 行了。 attlen 和其它字段可以用于解析表中一行内容。

47.8. `pg_authid`

`pg_authid` 包含有关数据库认证标识符(角色)的信息。 一个角色体现"用户"和"组"的概念。一个用户实际上只是一个设置了 `rolcanlogin` 标志的角色。任何角色(不管是否设置了 `rolcanlogin` 标志) 都可以有其它角色做为成员；参阅 `pg_auth_members` 。

因为这个系统表包含口令，所以它不是公共可读的。 `pg_roles` 是一个在 `pg_authid` 上的可读视图，只是把口令域填成了空白。

Chapter 20包含用户和权限管理的详细信息。

因为用户标识是集群范围的， `pg_authid` 在一个集群里所有的数据库之间是共享的：每个集群只有一个 `pg_authid` 拷贝， 而不是每个数据库一个。

Table 47-8. `pg_authid` 字段

名字	类型	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)
`rolname`	`name`	角色名称
`rolsuper`	`bool`	角色拥有超级用户权限
`rolinherit`	`bool`	角色自动继承其所属角色的权限
`rolcreaterole`	`bool`	角色可以创建更多角色
`rolcreatedb`	`bool`	角色可以创建数据库
`rolcatupdate`	`bool`	角色可以直接更新系统表。如果没有设置这个字段为真，即使超级用户也不能这么做。
`rolcanlogin`	`bool`	角色可以登录，也就是说，这个角色可以给予会话认证标识符。
`rolreplication`	`bool`	角色是一个复制的角色。也就是说，这个角色可以发起流复制（参阅Section 25.2.5）和使用 `pg_start_backup` 和 `pg_stop_backup` 设置/取消设置系统备份模式。
`rolconnlimit`	`int4`	对于可以登录的角色，限制其最大并发连接数量。-1 表示没有限制。
`rolpassword`	`text`	口令(可能是加密的)；如果没有则为 NULL。如果密码是加密的，该字段将以 `md5` 字符串开始，后面跟着一个32字符的十六进制MD5哈希值。该MD5哈希将是用户的口令加上用户名。 例如，如果用户 `joe` 的口令为 `xyzzy`，PostgreSQL 将存储MD5哈希为 `xyzzyjoe` 。不遵从这个格式的密码被假设为未加密的。
`rolvaliduntil`	`timestamptz`	口令失效时间(只用于口令认证)；如果没有失效期，则为 null

47.9. `pg_auth_members`

`pg_auth_members`显示角色之间的成员关系。任何非闭环的关系集合都是允许的。

因为用户标识是集群范围的， `pg_auth_members` 是在一个集群里的所有数据库之间共享的：每个集群里只有一个`pg_auth_members` 拷贝，而不是每个数据库一个。

Table 47-9. `pg_auth_members` 字段

名字	类型	引用	描述
`roleid`	`oid`	`pg_authid`.oid	拥有有成员的角色的ID
`member`	`oid`	`pg_authid` .oid	属于 `roleid` 角色的一个成员的角色的ID
`grantor`	`oid`	`pg_authid`.oid	赋予此成员关系的角色的 ID
`admin_option`	`bool`	如果 `member` 可以把 `roleid` 角色的成员关系赋予其它角色，则为真。	

47.10. `pg_cast`

`pg_cast` 表存储数据类型转换路径，包括内置路径和用户定义的路径。

应该注意， `pg_cast` 并不表示系统知道如何执行的每种类型转换； 只表示那些不能从某些通用法则中推导出的。例如，在一个域类型和它的基本类型之间的转换， 不是明确的由 `pg_cast` 表示的。另外一个重要的例外是 "automatic I/O conversion casts"， 这些使用数据类型自己的I/O函数转换为或从 `text` 或其他字符串类型转换的执行，不是由 `pg_cast` 明确表示的。

Table 47-10. `pg_cast` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`castsource`	`oid`	`pg_type` .oid	源数据类型的 OID
`casttarget`	`oid`	`pg_type` .oid	目标数据类型的 OID
`castfunc`	`oid`	`pg_proc` .oid	用于执行这个转换的函数的 OID 。如果转换方式不需要一个函数，那么为零。
`castcontext`	`char`	标识这个转换可以在什么环境里调用。 `e` 表示只能进行明确的转换 (使用 `CAST` 或 `::` 语法)。 `a` 表示在赋值给目标字段的时候隐含调用，也可以明确调用。 `i` 表示在表达式中隐含，当然也包括其它情况。	
`castmethod`	`char`	标识转换是怎么执行的。 `f` 表示使用了在 `castfunc` 字段里指定的函数。 `i` 表示使用了输入/输出函数。 `b` 表示该类型是二进制兼容的， 因此不需要什么转换。	

在 `pg_cast` 里列出的类型转换函数必须总是以类型转换的源类型作为它的第一个参数类型，并且返回类型转换的目的类型作为它的结果类型。一个类型转换函数最多有三个参数。如果出现了第二个参数， 必须是 `integer` 类型；它接受与目标类型关联的修饰词，如果没有，就是 -1 。如果出现了第三个参数， 那么必须是 `boolean` 类型；如果该类型转换是一种明确的转换，那么它接受 `true` ，否则接受 `false` 。

在 `pg_cast` 里创建一条源类型和目标类型相同的记录是合理的， 只要相关联的函数接受多过一个参数。这样的记录代表"长度转换函数"， 他们把该类型的数值转换为对特定的类型修饰词数值合法的值。

如果一条 `pg_cast` 记录有着不同的原类型和目标类型， 并且有一个接收多于一个参数的函数，那么它就意味着用一个步骤从一种类型转换到另外一种类型， 同时还附加一个长度转换。如果没有这样的记录，那么转换成一个使用了类型修饰词的类型涉及两个步骤， 一个是在数据类型之间转换，另外一个是附加修饰词。

47.11. `pg_class`

`pg_class` 表记载表和几乎所有有字段或者是那些类似表的东西。 包括索引(不过还要参阅 `pg_index`)、序列、视图、物化视图、 复合类型和一些特殊关系类型；参阅 `relkind` 。在下面， 当指所有这些对象的时候说"关系"(relations)。不是所有字段对所有关系类型都有意义。

Table 47-11. `pg_class` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`relname`	`name`	表、索引、视图等的名字。	
`relnamespace`	`oid`	`pg_namespace`.oid	包含这个关系的名字空间(模式)的OID
`reltype`	`oid`	`pg_type` .oid	如果有，则为对应这个表的行类型的数据类型的OID(索引为零，它们没有 `pg_type` 记录)。
`reloftype`	`oid`	`pg_type`.oid	对于类型表，为底层复合类型的OID，对于所有其他关系为0
`relowner`	`oid`	`pg_authid` .oid	关系所有者
`relam`	`oid`	`pg_am`.oid	如果行是索引，那么就是所用的访问模式(B-tree, hash 等等)
`relfilenode`	`oid`	这个关系在磁盘上的文件的名字，0表示这是一个"映射的"关系， 它的文件名取决于行级别的状态	
`reltablespace`	`oid`	`pg_tablespace`.oid	这个关系存储所在的表空间。如果为零，则意味着使用该数据库的缺省表空间。如果关系在磁盘

			上没有文件，则这个字段没有什么意义。
`relpages`	`int4`	以页(大小为 `BLCKSZ`)的此表在磁盘上的形式的大小。 它只是规划器用的一个近似值，是由 `VACUUM` , `ANALYZE` 和几个 DDL 命令，比如 `CREATE INDEX` 更新。	
`reltuples`	`float4`	表中行的数目。只是规划器使用的一个估计值，由 `VACUUM` , `ANALYZE` 和几个 DDL 命令，比如 `CREATE INDEX` 更新。	
`relallvisible`	`int4`	在表的可见映射中标记所有可见的页的数目。只是规划器使用的一个估计值，由 `VACUUM` , `ANALYZE` 和几个 DDL 命令，比如 `CREATE INDEX` 更新。	
`reltoastrelid`	`oid`	`pg_class` .oid	与此表关联的 TOAST 表的 OID ，如果没有为 0 。TOAST 表在一个从属表里"离线"存储大字段。
`reltoastidxid`	`oid`	`pg_class` .oid	对于 TOAST 表是它的索引的 OID ，如果不是 TOAST 表则为 0
`relhasindex`	`bool`	如果它是一个表而且至少有(或者最近有过)一个索引，则为真。	
`relisshared`	`bool`	如果该表在整个集群中由所有数据库共享则为真。只有某些系统表 (比如 `pg_database`)是共享的。	
`relpersistence`	`char`	`p` = permanent table（永久表）, `u` = unlogged table（未加载的表）, `t` = temporary table （临时表）	
`relkind`	`char`	`r` = ordinary table（普通表）, `i` = index（索引）, `S` = sequence（序列）, `v` = view（视图）, `m` = materialized view（物化视图）, `c` = composite type（复合类	

		型），t = TOAST table（TOAST 表），f = foreign table（外部表）
relnatts	int2	关系中用户字段数目(除了系统字段以外)。在 pg_attribute 里肯定有相同数目对应行。又见 pg_attribute.attnum 。
relchecks	int2	表里的 CHECK 约束的数目；参阅 pg_constraint 表
relhasoids	bool	如果为关系中每行都生成一个OID 则为真
relhaspkey	bool	如果这个表有一个(或者曾经有一个)主键，则为真。
relhasrules	bool	如表有（或曾经有）规则就为真；参阅 pg_rewrite 表
relhastriggers	bool	如果表有(或者曾经有)触发器，则为真；参阅 pg_trigger 表
relhassubclass	bool	如果有(或者曾经有)任何继承的子表，为真。
relispopulated	bool	如果关系是填充的则为真（对所有关系为真，除了一些物化视图）
relfrozenxid	xid	该表中所有在这个之前的事务ID 已经被一个固定的("frozen")事务 ID 替换。 这用于跟踪该表是否需要为了防止事务 ID 重叠或者允许收缩 pg_clog 而进行清理。如果该关系不是表则为零(InvalidTransactionId)。
relminmxid	xid	该表中所有在这个之前的多事务ID 已经被一个事务 ID 替换。 这用于跟踪该表是否需要为了防止多事务 ID 重叠或者允许收缩 pg_clog 而进行清理。如果该关系不是表则为零(InvalidTransactionId)。
relacl	aclitem[]	访问权限。参阅GRANT和REVOKE获取详细信息。
reloptions	text[]	访问方法特定的选项，使用"keyword=value"格式的字符串

`pg_class` 中的几个布尔标识是懒于维护的：如果这是正确的状态则他们被保证为真， 但是当状态不再是真时不会被立马重新设置为假。例如， `relhasindex` 是由 `CREATE INDEX` 设置的，但是从不用 `DROP INDEX` 删除。 相反，如果发现表没有索引了，由 `VACUUM` 清除 `relhasindex` 。 这个安排避免了竞态条件，提高了并发性。

47.12. `pg_event_trigger`

`pg_event_trigger` 表存储时间触发器。参阅Chapter 37 获取详细信息。

Table 47-12. `pg_event_trigger` 字段

名字	类型	引用	描述
`evtname`	`name`	触发器名字(必须是唯一的)	
`evtevent`	`name`	标识触发器触发的事件	
`evtowner`	`oid`	`pg_authid`.oid	事件触发器的所有者
`evtfoid`	`oid`	`pg_proc`.oid	要被调用的函数
`evtenabled`	`char`	控制在哪个session_replication_role模块中触发事件触发器。 `O` = 触发器在"origin" 和 "local"模块里面触发， `D` = 禁用触发器， `R` = 触发器在"replica"模块里面触发， `A` = 触发器总是触发。	
`evttags`	`text[]`	这个触发器将要触发的命令标签。如果为NULL，这个触发器的触发不受命令标签的基础限制。	

47.13. `pg_constraint`

`pg_constraint` 存储表上的检查约束、主键、唯一约束、外键约束和排除约束。 字段约束不会得到特殊对待。每个字段约束都等效于某些表约束。非空约束记录在 `pg_attribute` 表中。

未定义的约束触发器（用 `CREATE CONSTRAINT TRIGGER` 创建）也在此表中产生一个条目。

在域上面的检查约束也存储在这里。

Table 47-13. `pg_constraint` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`conname`	`name`	约束名(不一定是唯一的!)	
`connamespace`	`oid`	`pg_namespace`.oid	包含这个约束的名字空间的 OID
`contype`	`char`	`c` = 检查约束, `f` = 外键约束, `p` = 主键约束, `u` = 唯一约束, `t` = 约束触发器, `x` = 排除约束	
`condeferrable`	`bool`	这个约束可以推迟吗?	
`condeferred`	`bool`	缺省时这个约束是否推迟的?	
`convalidated`	`bool`	这个约束经过验证了吗？目前，外键约束和CHECK约束只能是假	
`conrelid`	`oid`	`pg_class` .oid	这个约束所在的表；如果不是表约束则为 0
`contypid`	`oid`	`pg_type` .oid	这个约束所在的域；如果不是一个域约束则为 0
`conindid`	`oid`	`pg_class` .oid	如果是唯一、主键、外键或排除约束，则为支持这个约束的索引；否则为0
`confrelid`	`oid`	`pg_class` .oid	如果是外键，则为参考的表；否则为 0
`confupdtype`	`char`	外键更新操作代码: `a` = 无动作, `r` = 限制, `c` =	

confupdtype	char	级联，n = 设置为空，d = 设置为缺省	
confdeltype	char	外键删除操作代码：a = 无动作，r = 限制，c = 级联，n = 设置为空，d = 设置为缺省	
confmatchtype	char	外键匹配类型：f = 全部，p = 部分，s = 简单的	
conislocal	bool	这个约束是为关系本地定义的。请注意，约束可以本地定义和同时继承。	
coninhcount	int4	这个约束直接继承祖先的数量。一个拥有非零祖先的约束不能被删除或重命名。	
connoinherit	bool	这个约束是为关系本地定义的。它是一个非继承的约束。	
conkey	int2[]	pg_attribute .attnum	如果是表约束（包含外键，但是不包含约束触发器），则是约束字段的列表
confkey	int2[]	pg_attribute .attnum	如果是一个外键，是参考的字段的列表
conpfeqop	oid[]	pg_operator .oid	如果是一个外键，是PK = FK比较的相等操作符的列表
conppeqop	oid[]	pg_operator .oid	如果是一个外键，是PK = PK比较的相等操作符的列表
conffeqop	oid[]	pg_operator .oid	如果是一个外键，是FK = FK比较的相等操作符的列表
conexclop	oid[]	pg_operator .oid	如果是一个排除约束，是每个字段排除操作符的列表
conbin	pg_node_tree	如果是一个检查约束，那就是其表达式的内部形式	
consrc	text	如果是检查约束，则是表达式的人类可读形式	

> **Note:** `consrc` 在被引用的对象改变之后不会被更新，它不会跟踪字段的名字修改。 与其依赖这个字段，最好还是使用 `pg_get_constraintdef()` 来抽取一个检查约束的定义。
>
> **Note:** `pg_class.relchecks` 需要和在此表上为每个关系找到的检查约束的数目一致。

47.14. pg_collation

`pg_collation` 表描述可用的排序规则，本质上从一个SQL名字映射到操作系统本地类别。 参阅Section 22.2获取详细信息。

Table 47-14. `pg_collation` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`collname`	`name`	排序规则名 (每个名字空间和编码唯一)	
`collnamespace`	`oid`	`pg_namespace`.oid	包含这个排序规则的名字空间的OID
`collowner`	`oid`	`pg_authid` .oid	排序规则的所有者
`collencoding`	`int4`	排序规则可用的编码， 如果适用于任意编码为-1	
`collcollate`	`name`	这个排序规则对象的 `LC_COLLATE`	
`collctype`	`name`	这个排序规则对象的 `LC_CTYPE`	

请注意这个表中的主键是(`collname` , `collencoding` , `collnamespace`)不只是(`collname` , `collnamespace`)。 PostgreSQL通常忽略所有的 `collencoding` 不等于当前数据库编码或-1的排序规则，并且和 `collencoding` = -1 里的条目有相同名字的新条目的创建是被禁止的。因此，使用一个受限制的SQL名字 (`_schema_` . `_name_`)足够去定义一个排序规则，即使根据表的定义这不是唯一的。 这种方式定义表的原因是initdb在集群初始化时用所有在系统上可用的区域设置的条目填充了它， 所以必须能够保持所有可能在集群中用的到编码的条目。

在 `template0` 数据库中，创建编码不匹配数据库编码的排序规则可能是有用的， 因为他们可以匹配稍后从 `template0` 复制来的数据库编码。目前这些必须手动完成。

47.15. pg_conversion

pg_conversion 描述编码转换信息。参阅 CREATE CONVERSION获取更多信息。

Table 47-15. pg_conversion 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
conname	name	转换名字(在一个名字空间里是唯一的)	
connamespace	oid	pg_namespace.oid	包含这个转换的名字空间的 OID
conowner	oid	pg_authid .oid	编码转换的属主
conforencoding	int4	源编码 ID	
contoencoding	int4	目的编码 ID	
conproc	regproc	pg_proc.oid	转换过程
condefault	bool	如果这是缺省转换则为真	

47.16. pg_database

pg_database 表存储关于可用数据库的信息。数据库是用 CREATE DATABASE创建的。参考Chapter 21 获取一些参数的详细含义。

和大多数系统表不同, pg_database 是在一个集群里的所有数据库共享的： 每个集群只有一份 pg_database 拷贝， 而不是每个数据库一份。

Table 47-16. pg_database 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
datname	name	数据库名字	
datdba	oid	pg_authid .oid	数据库所有人，通常为其创建者
encoding	int4	数据库的字符编码方式(pg_encoding_to_char() 能够将这个数字转换为相应的编码名称)	
datcollate	name	这个数据库的LC_COLLATE	
datctype	name	这个数据库的LC_CTYPE	
datistemplate	bool	如果为真则此数据库可以用于 CREATE DATABASE 的 TEMPLATE 子句，把新数据库创建为此数据库的克隆。	
datallowconn	bool	如果为假则没有人可以连接到这个数据库。这个字段用于保护 template0 数据库不被更改。	
datconnlimit	int4	设置该数据库上允许的最大并发连接数，-1 表示无限制。	
datlastsysoid	oid	数据库里最后一个系统 OID ；对pg_dump特别有用。	
datfrozenxid	xid	该数据库中中所有在这个之前的事务 ID 已经被一个固定的("frozen") 事务 ID 替换。这用于跟踪该数据库是否需要为了防止事务 ID 重	

`datfrozenxid`	`xid`	叠或者允许收缩 `pg_clog` 而进行清理。它是针对每个表的 `pg_class` . `relfrozenxid` 中的最小值。	
datminmxid	xid	该数据库中中所有在这个之前的多事务 ID 已经被一个事务 ID 替换。 这用于跟踪该数据库是否需要为了防止事务 ID 重叠或者允许收缩 pg_clog 而进行清理。它是针对每个表的 pg_class . relfrozenxid 中的最小值。	
`dattablespace`	`oid`	`pg_tablespace`.oid	该数据库的缺省表空间。这个数据库里，所有 `pg_class` . `reltablespa` 为零的表都将保存在这个空间里；特别要指出的是，所有非共享的系统表也都放在这里。
datacl	aclitem[]	访问权限，参阅GRANT和REVOKE获取详细信息。	

47.17. `pg_db_role_setting`

`pg_db_role_setting` 记录已经为运行时配置变量设置的缺省值， 为每个角色和数据库的组合。

不像大多数系统表， `pg_db_role_setting` 在一个集群中的所有数据库中共享： 这只是每个集群 `pg_db_role_setting` 的一个复制，不是每个数据库。

Table 47-17. `pg_db_role_setting` 字段

名字	类型	引用	描述
`setdatabase`	`oid`	`pg_database`.oid	设置适用于的数据库的OID,如果不是特定于数据库的则为0
`setrole`	`oid`	`pg_authid` .oid	设置适用于的角色的OID,如果不是特定于角色的则为0
`setconfig`	`text[]`	默认为运行时配置变量	

47.18. `pg_default_acl`

`pg_default_acl`表存储分配给新创建对象的初始化权限。

Table 47-18. `pg_default_acl` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`defaclrole`	`oid`	`pg_authid` .oid	与该条目相关的角色的OID
`defaclnamespace`	`oid`	`pg_namespace`.oid	与这个条目相关的名字空间的OID，如果没有则为0
`defaclobjtype`	`char`	这个条目的对象的类型： `r` = 关系（表，视图）， `S` = 序列， `f` = 函数， `T` = 类型	
`defaclacl`	`aclitem[]`	这种类型的对象在创建时应该有的访问权限	

`pg_default_acl`条目显示了分配给属于指定用户的对象的初始权限。 当前有两种条目类型： `defaclnamespace` = 0的"全局"条目， 和引用一个特定模式的"每模式"条目。如果当前是全局条目， 那么它为每个对象类型重写正常硬链接的缺省权限。 如果当前是每模式条目，表示权限被添加到全局或硬链接的缺省权限。

请注意，当一个ACL条目在另一个表中为空时，用来为它的对象表示硬链接的缺省权限， 不是此刻可能在`pg_default_acl`中的东西。 `pg_default_acl`只在对象创建时访问。

47.19. pg_depend

`pg_depend` 表记录数据库对象之间的依赖关系。 这个信息允许 `DROP` 命令找出哪些其它对象必须由 `DROP CASCADE` 删除， 或者是在 `DROP RESTRICT` 的情况下避免删除。

这个表的功能类似 `pg_shdepend` ， 用于记录那些在数据库集群之间共享的对象之间的依赖性关系。

Table 47-19. `pg_depend` 字段

名字	类型	引用	描述
`classid`	`oid`	`pg_class` .oid	有倚赖对象所在系统表的OID
`objid`	`oid`	任意 OID 属性	指定的依赖对象的OID
`objsubid`	`int4`	对于表字段，这个是该属性的字段数(`objid` 和 `classid` 引用表本身)。对于所有其它对象类型，目前这个字段是零。	
`refclassid`	`oid`	`pg_class` .oid	被引用对象所在的系统表的OID
`refobjid`	`oid`	任意 OID 属性	指定的被引用对象的 OID
`refobjsubid`	`int4`	对于表字段，这个是该字段的字段号(`refobjid` 和 `refclassid` 引用表本身)。对于所有其它对象类型，目前这个字段是零。	
`deptype`	`char`	一个定义这个依赖关系特定语义的代码。见下文。	

在所有情况下，一个 `pg_depend` 记录表示被引用的对象不能在有依赖的对象被删除前删除。 不过，这里还有几种由 `deptype` 定义的情况：

`DEPENDENCY_NORMAL` (`n`)

独立创建的对象之间的一般关系。有倚赖的对象可以在不影响被引用对象的情况下删除。 被引用对象只有在声明了 `CASCADE` 的情况下删除，这时有依赖的对象也被删除。 例子：一个表字段对其数据类型有一般依赖关系。

`DEPENDENCY_AUTO` (`a`)

有依赖对象可以和被引用对象分别删除，并且如果删除了被引用对象则应该被自动删除 (不管是 `RESTRICT` 或 `CASCADE` 模式)。例子： 一个表上面的命名约束是在该表上的自动依赖关系，因此如果删除了表，它也会被删除。

`DEPENDENCY_INTERNAL` (`i`)

有依赖的对象是作为被引用对象的一部分创建的，实际上只是它的内部实现的一部分。 `DROP` 有依赖对象是不能直接允许的(将告诉用户发出一条删除被引用对象的 `DROP`)。 一个对被引用对象的 `DROP` 将传播到有依赖对象，不管是否声明了 `CASCADE` 。 例子：一个创建来强制外键约束的触发器在该约束的 `pg_constraint` 记录上是标记为内部依赖的。

`DEPENDENCY_EXTENSION` (`e`)

依赖对象是被依赖对象*extension*的一个成员（参阅 `pg_extension` ）。 依赖对象只可以通过在被依赖对象上 `DROP EXTENSION` 删除。 函数上这个依赖类型和内部依赖一样动作，但是它为了清晰和简化pg_dump保持分开。

`DEPENDENCY_PIN` (`p`)

没有有依赖对象；这种类型的记录标志着系统本身依赖于被引用对象，因此这个对象决不能被删除。 这种类型的记录只有在 `initdb` 的时候创建。有依赖对象的字段里是零。

将来可能还会有其它依赖的风格。

47.20. pg_description

pg_description 表可以给每个数据库对象存储一个可选的描述(注释)。 你可以用COMMENT 命令操作这些描述，并且可以用psql 的 \d 命令查看。许多内置的系统对象的描述提供了 pg_description 的初始内容。

pg_shdescription 提供了类似的功能，它记录了整个集群范围内共享对象的注释。

Table 47-20. pg_description 字段

名字	类型	引用	描述
objoid	oid	任意 oid 属性	这条描述所描述的对象的 OID
classoid	oid	pg_class .oid	这个对象出现的系统表的 OID
objsubid	int4	对于一个表字段的注释，它是字段号(objoid 和 classoid 指向表自身)。对于其它对象类型，它是零。	
description	text	作为对该对象的描述的任意文本	

47.21. pg_enum

pg_enum 表包含显示每个枚举类型值和标签的记录。 给定枚举类型的内部表示实际上是 pg_enum 里面相关行的OID。

Table 47-21. pg_enum 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
enumtypid	oid	pg_type .oid	拥有这个枚举值的 pg_type 记录的OID
enumsortorder	float4	这个枚举值在它的枚举类型中的排序位置	
enumlabel	name	这个枚举值的文本标签	

pg_enum 行的OID跟着一个特殊规则：偶数的OID保证用和它们的枚举类型一样的排序顺序排序。 也就是，如果两个偶数OID属于相同的枚举类型，那么较小的OID必须有较小 enumsortorder 值。 奇数OID需要毫无关系的排序顺序。这个规则允许枚举比较例程在许多常见情况下避开目录查找。 创建和修改枚举类型的例程只要可能就尝试分配偶数OID给枚举值。

当创建了一个枚举类型时，它的成员赋予了排序顺序位置 1.. _n_ 。 但是随后添加的成员可能会分配 enumsortorder 的负值或分数值。 对这些值的唯一要求是它们要正确的排序和在每个枚举类型中唯一。

47.22. pg_extension

pg_extension 表存储了关于已安装的扩展的信息。 参阅Section 35.15获取有关扩展的详细信息。

Table 47-22. pg_extension 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
extname	name	扩展名	
extowner	oid	pg_authid .oid	扩展的所有者
extnamespace	oid	pg_namespace .oid	包含扩展的输出对象的模式
extrelocatable	bool	如果扩展可以重新加载到另一个模式则为真	
extversion	text	扩展的版本名	
extconfig	oid[]	pg_class .oid	扩展的配置表的 regclass OID 的数组， 如果没有则为 NULL
extcondition	text[]	扩展的配置表的 WHERE 子句过滤条件的数组， 如果没有则为 NULL	

请注意，不同于大多数有"namespace"字段的表， extnamespace 并不意味着扩展属于哪个模式。扩展名从不模式限定。 extnamespace 表明模式包含大多数或所有的扩展的对象。如果 extrelocatable 为真， 那么这个模式必须实际上包含所有属于该扩展的模式限定的对象。

47.23. pg_foreign_data_wrapper

pg_foreign_data_wrapper 表存储外部数据封装器定义。 一个外部数据封装器是在外部服务器上驻留外部数据的机制，是可以访问的。

Table 47-23. pg_foreign_data_wrapper 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
fdwname	name	外部数据封装器名	
fdwowner	oid	pg_authid .oid	外部数据封装器的所有者
fdwhandler	oid	pg_proc .oid	引用一个负责为外部数据封装器提供扩展例程的处理函数。如果没有提供处理函数则为零
fdwvalidator	oid	pg_proc .oid	引用一个验证器函数，这个验证器函数负责验证给予外部数据封装器的选项、 外部服务器选项和使用外部数据封装器的用户映射的有效性。 如果没有提供验证器函数则为零。
fdwacl	aclitem[]	访问权限；参阅GRANT和REVOKE获取详细信息。	
fdwoptions	text[]	外部数据封装器指定选项，使用"keyword=value"格式的字符串	

47.24. `pg_foreign_server`

`pg_foreign_server` 表存储外部服务器定义。 一个外部服务器描述了一个外部数据源，例如一个远程服务器。 外部服务器通过外部数据封装器访问。

Table 47-24. `pg_foreign_server` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`srvname`	`name`	外部服务器名	
`srvowner`	`oid`	`pg_authid`.oid	外部服务器的所有者
`srvfdw`	`oid`	`pg_foreign_data_wrapper`.oid	这个外部服务器的外部数据封装器的OID
`srvtype`	`text`	服务器的类型（可选）	
`srvversion`	`text`	服务器的版本（可选）	
`srvacl`	`aclitem[]`	访问权限；参阅GRANT和REVOKE获取详细信息。	
`srvoptions`	`text[]`	外部服务器指定选项，使用"keyword=value"格式的字符串。	

47.25. `pg_foreign_table`

`pg_foreign_table`表包含关于外表的辅助信息。 一个外表首先通过一个`pg_class`记录表现，就像一个普通表。 它的`pg_foreign_table`记录包含只和外表相关的信息，没有任何其他类型的关系。

Table 47-25. `pg_foreign_table` 字段

名字	类型	引用	描述
`ftrelid`	`oid`	`pg_class`.oid	这个外表的`pg_class`记录的OID
`ftserver`	`oid`	`pg_foreign_server`.oid	这个外表的外部服务器的OID
`ftoptions`	`text[]`	外表选项，使用"keyword=value"格式的字符串	

47.26. `pg_index`

`pg_index` 包含关于索引的一部分信息。其它的信息大多数在 `pg_class` 。

Table 47-26. `pg_index` 字段

名字	类型	引用	描述
`indexrelid`	`oid`	`pg_class` .oid	这个索引在 `pg_class` 里的记录的 OID
`indrelid`	`oid`	`pg_class` .oid	使用这个索引的表在 `pg_class` 里的记录的 OID
`indnatts`	`int2`	索引中的字段数目(复制的 `pg_class.relnatts`)	
`indisunique`	`bool`	如果为真，这是个唯一索引	
`indisprimary`	`bool`	如果为真，该索引代表该表的主键。这个字段为真的时候 `indisunique` 应该总是为真。	
`indisexclusion`	`bool`	如果为真，那么这个索引支持一个排除约束	
`indimmediate`	`bool`	如果为真，立即强制对插入进行唯一性检查(如果 `indisunique` 不为真则是不相关的)	
`indisclustered`	`bool`	如果为真，那么该表最后在这个索引上建了簇。	
`indisvalid`	`bool`	如果为真，那么该索引可以用于查询。如果为假，那么该索引可能不完整：仍然必须在 `INSERT` / `UPDATE` 操作时进行更新，不过不能安全的用于查询。 如果是唯一索引，那么唯一属性也	

		不保证为真。	
indcheckxmin	bool	如果为真，查询必须不是使用索引，知道这个 pg_index 行的 xmin 低于它们的 TransactionXmin 事件地平线，因为该表可能包含他们能看到的不兼容的行的断热链。	
indisready	bool	如果为真，该索引目前已准备好插入。如果为假，那么该索引必须通过 INSERT / UPDATE 操作忽略。	
indislive	bool	如果为假，那么该索引正在被删除，并且应该被所有目的所忽略（包括热安全决策）	
indkey	int2vector	pg_attribute .attnum	这是一个包含 indnatts 值的数组，这些数组值表示这个索引所建立的表字段。 比如一个值为 1 3 的意思是第一个字段和第三个字段组成这个索引键字。 这个数组里的零表明对应的索引属性是在这个表字段上的一个表达式，而不是一个简单的字段引用。
indcollation	oidvector	pg_collation .oid	对于每个在这个索引键字中的字段，这个字段包含用于这个索引的排序规则的OID。
indclass	oidvector	pg_opclass .oid	对于索引键字里面的每个字段，这个字段都包含一个指向所使用的操作符类的 OID， 参阅 pg_opclass 获取细节。
indoption	int2vector	这是 indnatts 值的一个数组，存储每个字段标志位。 这个标志位的含义是通过索引的访问方式定义的。	

`indexprs`	`pg_node_tree`	表达式树(以 `nodeToString()` 形式表现)用于那些非简单字段引用的索引属性。 它是一个列表，在 `indkey` 里面的每个零条目一个元素。 如果所有索引属性都是简单的引用，则为空。
`indpred`	`pg_node_tree`	部分索引断言的表达式树(以 `nodeToString()` 的形式表现)。 如果不是部分索引，则是空字符串。

47.27. `pg_inherits`

`pg_inherits`记录关于表继承层次的信息。数据库里每个直接的子系表都有一条记录。 间接的继承可以通过追溯记录链来判断。

Table 47-27. `pg_inherits` 字段

名字	类型	引用	描述
`inhrelid`	`oid`	`pg_class`.oid	子表的OID
`inhparent`	`oid`	`pg_class`.oid	父表的OID
`inhseqno`	`int4`	如果一个子表存在多个直系父表(多重继承)，这个数字表明此继承字段的排列顺序。 计数从 1 开始。	

47.28. `pg_language`

`pg_language` 登记编程语言，你可以用这些语言或接口写函数或者存储过程。 参阅CREATE LANGUAGE和Chapter 39获取更多关于语言处理器的信息。

Table 47-28. `pg_language` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`lanname`	`name`	语言的名字	
`lanowner`	`oid`	`pg_authid`.oid	语言的所有者
`lanispl`	`bool`	对于内部语言而言是假(比如SQL)， 对于用户定义的语言则是真。 目前，pg_dump仍然使用这个东西判断哪种语言需要转储， 但是这些可能在将来被其它机制取代。	
`lanpltrusted`	`bool`	如果这是可信语言则为真， 意味着系统相信它不会被授予任何正常 SQL 执行环境之外的权限。 只有超级用户可以用不可信的语言创建函数。	
`lanplcallfoid`	`oid`	`pg_proc` .oid	对于非内部语言， 这是指向该语言处理器的引用， 语言处理器是一个特殊函数， 负责执行以某种语言写的所有函数。
`laninline`	`oid`	`pg_proc` .oid	这个字段引用一个负责执行"inline"匿名代码块的函数（DO块）。 如果不支持内联块则为零。
`lanvalidator`	`oid`	`pg_proc` .oid	这个字段引用一个语言校验器函数， 它负责检查新创建的函数的语法和有效性。 如果没有提供校验器， 则为零。
`lanacl`	`aclitem[]`	访问权限， 参阅GRANT和REVOKE获取细节。	

47.29. pg_largeobject

pg_largeobject 表保存那些标记着"大对象"的数据。 一个大对象是使用其创建时分配的 OID 标识的。每个大对象都分解成足够小的小段或者 "页面"以便以行的形式存储在 pg_largeobject 里。 每页的数据定义为 LOBLKSIZE (目前是 BLCKSZ/4 或者通常是 2K 字节)。

PostgreSQL 9.0之前，没有许可结构与大对象相关。因此， pg_largeobject 公开可读并且可以用来包含系统中所有大对象的OID（和内容）。现在不再是这样了； 使用 pg_largeobject_metadata 获取大对象OID的列表。

Table 47-29. pg_largeobject 字段

名字	类型	引用	描述
loid	oid	pg_largeobject_metadata.oid	包含本页的大对象的标识符
pageno	int4	本页在其大对象数据中的页码从零开始计算	
data	bytea	存储在大对象中的实际数据。这些数据绝不会超过 LOBLKSIZE 字节，而且可能更少。	

pg_largeobject 的每一行保存一个大对象的一个页面， 从该对象内部的字节偏移 (pageno * LOBLKSIZE)开始。这种实现允许松散的存储： 页面可以丢失，而且可以比 LOBLKSIZE 字节少(即使它们不是对象的最后一页)。 大对象内丢失的部分读做零。

47.30. `pg_largeobject_metadata`

`pg_largeobject_metadata`表存储与大数据相关的元数据。实际的大对象数据存储在`pg_largeobject`里。

Table 47-30. `pg_largeobject_metadata` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`lomowner`	`oid`	`pg_authid`.oid	大对象的所有者
`lomacl`	`aclitem[]`	访问权限；参阅GRANT和REVOKE获取详细信息。	

47.31. pg_namespace

pg_namespace 存储名字空间。名字空间是 SQL 模式下层的结构：每个名字空间有独立的关系， 类型等集合但并不会相互冲突。

Table 47-31. pg_namespace 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
nspname	name	名字空间的名字	
nspowner	oid	pg_authid.oid	名字空间的所有者
nspacl	aclitem[]	访问权限；参阅GRANT和REVOKE获取细节。	

47.32. `pg_opclass`

`pg_opclass`定义索引访问方法操作符类。 每个操作符类为一种特定数据类型和一种特定索引访问方法定义索引字段的语义。 一个操作符类本质上指定一个特定的操作符族适用于一个特定的可索引的字段数据类型。 索引的字段实际可用的族中的操作符集是接受字段的数据类型作为它们的左边的输入的那个。

操作符类在Section 35.14里有比较详细的描述。

Table 47-32. `pg_opclass` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`opcmethod`	`oid`	`pg_am` .oid	操作符类所服务的索引访问方法
`opcname`	`name`	这个操作符类的名字	
`opcnamespace`	`oid`	`pg_namespace` .oid	这个操作符类的名字空间
`opcowner`	`oid`	`pg_authid` .oid	操作符类属主
`opcfamily`	`oid`	`pg_opfamily` .oid	包含该操作符类的操作符族
`opcintype`	`oid`	`pg_type` .oid	操作符类索引的数据类型
`opcdefault`	`bool`	如果操作符类是 `opcintype` 的缺省，则为真	
`opckeytype`	`oid`	`pg_type` .oid	索引数据的类型，如果和 `opcintype` 相同则为零

一个操作符类的 `opcmethod` 必须匹配包含它的操作符族的 `opfmethod` 。 同样，对于任意给定的 `opcmethod` 和 `opcintype` 的组合， 不能有超过一个 `pg_opclass` 行有 `opcdefault` 为真。

47.33. `pg_operator`

`pg_operator` 存储有关操作符的信息。参阅 CREATE OPERATOR 和 节33.12 获取这些操作符参数的细节。

Table 47-33. `pg_operator` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`oprname`	`name`	操作符的名字	
`oprnamespace`	`oid`	`pg_namespace` .oid	包含此操作符的名字空间的 OID
`oprowner`	`oid`	`pg_authid` .oid	操作符所有者
`oprkind`	`char`	`b` = infix = 中缀("两边"), `l` = 前缀("左边"), `r` = 后缀("右边")	
`oprcanmerge`	`bool`	这个操作符支持合并连接	
`oprcanhash`	`bool`	这个操作符支持 Hash 连接	
`oprleft`	`oid`	`pg_type` .oid	左操作数的类型
`oprright`	`oid`	`pg_type` .oid	右操作数的类型
`oprresult`	`oid`	`pg_type` .oid	结果类型
`oprcom`	`oid`	`pg_operator` .oid	此操作符的交换符，如果存在的话
`oprnegate`	`oid`	`pg_operator` .oid	此操作符的反转器，如果存在的话
`oprcode`	`regproc`	`pg_proc` .oid	实现这个操作符的函数
`oprrest`	`regproc`	`pg_proc` .oid	此操作符的约束选择性计算函数
`oprjoin`	`regproc`	`pg_proc` .oid	此操作符的连接选择性计算函数

未用的字段包含零。比如， `oprleft` 对于前缀操作符而言是零。

47.34. pg_opfamily

pg_opfamily 表定义操作符族。每个操作符族是一个操作符和相关支持例程的集合， 其中的例程实现为一个特定的索引访问方式指定的语义。 另外，族中的操作符都是"兼容的"， 通过由访问方式指定的方法。操作符族的概念允许交叉数据类型操作符和索引一起使用， 并且合理的使用访问方式的语义的知识。

操作符族在Section 35.14里面描述。

Table 47-34. pg_opfamily 字段

名字	类型	引用	描述
oid	oid	行标识符(隐藏属性; 必须明确选择)	
opfmethod	oid	pg_am .oid	操作符族使用的索引方法
opfname	name	这个操作符族的名字	
opfnamespace	oid	pg_namespace .oid	这个操作符的名字空间
opfowner	oid	pg_authid .oid	操作符族的所有者

定义一个操作符族的大多数信息不在它的 pg_opfamily 行里面， 而是在相关的行 pg_amop ， pg_amproc 和 pg_opclass 里。

47.35. `pg_pltemplate`

`pg_pltemplate` 表为过程语言存储"模板"信息。 一个语言的模板允许该语言可以在某个数据库里使用简单的 `CREATE LANGUAGE` 命令创建， 而不需要声明实现细节。

和许多系统表不一样， `pg_pltemplate` 是在集群里的所有数据库之间共享的： 每个集群只有一个 `pg_pltemplate` 的拷贝，而不是每个数据库一个。 这样就允许这些信息在需要时每个数据库都可以访问。

Table 47-35. `pg_pltemplate` 字段

名字	类型	描述
`tmplname`	`name`	这个模板所应用的语言的名字
`tmpltrusted`	`boolean`	如果语言被认为是可信的，则为真
`tmpldbacreate`	`boolean`	如果语言可以通过数据库所有者创建则为真
`tmplhandler`	`text`	调用处理器函数的名字
`tmplinline`	`text`	匿名块处理器函数的名字，如果没有则为null
`tmplvalidator`	`text`	校验函数的名字，如果没有则为 NULL
`tmpllibrary`	`text`	实现语言的共享库的路径
`tmplacl`	`aclitem[]`	模板的访问权限(未使用)

目前还没有任何命令可以用于操作过程语言模板；要修改内置的信息， 超级用户必须使用普通的 `INSERT`， `DELETE`， `UPDATE` 命令修改该表。

> **Note:** 很有可能在将来的PostgreSQL版本中删除 `pg_pltemplate`， 为了支持过程语言的信息保持在它们各自的扩展安装脚本中。

47.36. pg_proc

`pg_proc` 表存储关于函数(或过程)的信息。参阅CREATE FUNCTION 和Section 35.3获取更多信息。

该表包含聚集函数和普通函数的数据。如果 `proisagg` 为真， 那么在 `pg_aggregate` 里应该有一个匹配行。

Table 47-36. `pg_proc` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`proname`	`name`	函数名字	
`pronamespace`	`oid`	`pg_namespace`.oid	包含该函数名字空间的 OID
`proowner`	`oid`	`pg_authid`.oid	函数的所有者
`prolang`	`oid`	`pg_language`.oid	这个函数的实现语言或调用接口
`procost`	`float4`	估计执行成本(以cpu_operator_cost为单位); 如果 `proretset`，这是每一行返回的成本	
`prorows`	`float4`	估计的结果行数(如果非 `proretset` 则为零)	
`provariadic`	`oid`	`pg_type`.oid	可变数组参数的元素的数据类型，如果函数没有可变参数则为零
`protransform`	`regproc`	`pg_proc`.oid	调用这个函数可以简化其他函数(参阅Section 35.9.11)
`proisagg`	`bool`	函数是聚集函数	
`proiswindow`	`bool`	函数是窗口函数	
`prosecdef`	`bool`	函数是一个安全定义器(也就是一个"setuid"函数)	
		该函数没有副作用。没有关于该参数的信息传递，除非	

`proleakproof`	`bool`	通过返回值。 任何函数都有可能抛出一个错误，取决于它的参数值是不加密的。	
`proisstrict`	`bool`	如果任何调用参数是空，那么函数返回空。这时函数实际上连调用都不调用。 不是"strict"的函数必须准备处理空输入。	
`proretset`	`bool`	函数返回一个集合(也就是说，指定数据类型的多个数值)	
`provolatile`	`char`	告诉该函数的结果是否只依赖于它的输入参数，或者还会被外接因素影响。 对于"不可变的"(immutable)函数它是 `i`， 这样的函数对于相同的输入总是产生相同的结果。对于"稳定的"(stable) 函数它是 `s`，(对于固定输入)其结果在一次扫描里不变。 对于"易变"(volatile)函数它是 `v`，其结果可能在任何时候变化。 `v` 也用于那些有副作用的函数，因此调用它们无法得到优化。	
`pronargs`	`int2`	参数数目	
`pronargdefaults`	`int2`	有缺省值的参数的数量	
`prorettype`	`oid`	`pg_type` .oid	返回值的数据类型
`proargtypes`	`oidvector`	`pg_type` .oid	一个存放函数参数的数据类型的数组。数组里只包括输入参数(包括 `INOUT` 和 `VARIADIC` 参数)，因此代表该函数的调用签名(接口)。
			一个包含函数参数的数据类型的数组。数组里包括所有参数的类型(包括 `OUT` 和 `INOUT` 参数)；不过，如果所有参数都是 `IN` 参

			数，那么这个字段就会是空。请注意数组下标是以 1 为起点的，而因为历史原因，`proargtypes` 的下标起点为 0 。
proargmodes	char[]	一个保存函数参数模式的数组，编码如下： `i` 表示 `IN` 参数， `o` 表示 `OUT` 参数， `b` 表示 `INOUT` 参数， `v` 表示 `VARIADIC` 参数， `t` 表示 `TABLE` 参数。 如果所有参数都是 `IN` 参数，那么这个字段为空。请注意， 下标对应的是 `proallargtypes` 的位置，而不是 `proargtypes` 。	
`proargnames`	`text[]`	一个保存函数参数的名字的数组。没有名字的参数在数组里设置为空字符串。 如果没有一个参数有名字，这个字段将是空。请注意，此数组的下标对应 `proallargtypes` 而不是 `proargtypes` 。	
proargdefaults	pg_node_tree	缺省值的表达式树（用 `nodeToString()` 表示）。 是和 `pronargdefaults` 参数一起列出的，对应最后 _N_ 个输入 参数（也就是，最后 _N_ 个 `proargtypes` 位置）。 如果没有有缺省的参数，那么这个字段为null。	
`prosrc`	`text`	这个字段告诉函数处理器如何调用该函数。它实际上对于解释语言来说就是函数的源程序， 或者一个链接符号，一个文件名，或者是任何其它的东西，具体取决于语言/调用习惯的实现。	
probin	text	关于如何调用该函数的附加信息。同样，其含义也是和语言相关的。	
`proconfig`	`text[]`	函数的运行时配置变量的本地设置	

proacl	aclitem[]	访问权限；参阅GRANT和REVOKE获取细节。

对于内置和动态加载得编译函数， `prosrc` 包含函数的 C 语言名字(链接符号)。 对于所有其它当前已知的语言类型， `prosrc` 包含该函数的源文本。 `probin` 除了用于动态加载的 C 函数之外没有其它用途， 这个时候它给出包含给函数的共享库的文件名。

47.37. pg_range

pg_range 存储关于范围类型的信息。除了 pg_type 里类型的记录。

Table 47-37. pg_range 字段

名字	类型	引用	描述
rngtypid	oid	pg_type .oid	范围类型的OID
rngsubtype	oid	pg_type .oid	这个范围类型的元素类型（子类型）的OID
rngcollation	oid	pg_collation .oid	用于范围比较的排序规则的OID，如果没有则为零
rngsubopc	oid	pg_opclass .oid	用于范围比较的子类型的操作符类的OID
rngcanonical	regproc	pg_proc .oid	转换范围类型为规范格式的函数的OID，如果没有则为0。
rngsubdiff	regproc	pg_proc .oid	返回两个 double precision 元素值的不同的函数的OID，如果没有则为0。

rngsubopc （如果元素类型是可排序的则加上 rngcollation ） 决定用于范围类型的排序顺序。当元素类型是离散的时使用 rngcanonical 。 rngsubdiff 是可选的，但是应该应用于提高范围类型上的GiST索引的性能。

47.38. `pg_rewrite`

`pg_rewrite` 存储为表和视图定义的重写规则。

Table 47-38. `pg_rewrite` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`rulename`	`name`	规则名称	
`ev_class`	`oid`	`pg_class`.oid	使用这条规则的表名称
`ev_attr`	`int2`	这条规则适用的字段(目前总是为-1，表示整个表)	
`ev_type`	`char`	规则适用的事件类型：1 = `SELECT`, 2 = `UPDATE`, 3 = `INSERT`, 4 = `DELETE`	
`ev_enabled`	`char`	控制规则在哪个session_replication_role模块触发。 `O` = 规则 "origin" 和 "local" 模块触发, `D` = 规则被禁用, `R` = 规则在 "replica" 模块触发, `A` = 规则总是触发。	
`is_instead`	`bool`	如果该规则是 `INSTEAD` 规则，那么为真	
`ev_qual`	`pg_node_tree`	规则的资格条件的表达式树(以 `nodeToString()` 形式存在)	
`ev_action`	`pg_node_tree`	规则动作的查询树(以 `nodeToString()` 形式存在)	

> **Note:** 如果一个表在这个系统表里有任何规则存在，那么 `pg_class.relhasrules` 必须为真。

47.39. `pg_seclabel`

`pg_seclabel` 表存储数据对象上的安全标签。 安全标签可以和SECURITY LABEL命令一起操作。 查看安全标签的简单些的方法请参阅Section 47.65。

`pg_shseclabel` 表的作用类似， 只是它是用于在一个数据库集群内共享的数据库对象的安全标签上的。

Table 47-39. `pg_seclabel` 字段

名字	类型	引用	描述
`objoid`	`oid`	任意OID属性	这个安全标签所属的对象的OID
`classoid`	`oid`	`pg_class` .oid	出现这个对象的系统目录的OID
`objsubid`	`int4`	对于一个在表字段上的安全标签，这是字段号（ `objoid` 和 `classoid` 参考表本身）。对于所有其他对象类型，这个字段是零。	
`provider`	`text`	与这个标签相关的标签提供程序。	
`label`	`text`	应用于这个对象的安全标签。	

47.40. pg_shdepend

`pg_shdepend` 记录数据库对象和共享对象(比如角色)之间的依赖性关系。 这些信息允许PostgreSQL保证在企图删除这些对象之前， 这些对象是没有被引用的。

`pg_depend` 表的作用类似， 只是它是用于在一个数据库内部的对象的依赖性关系的。

和其它大多数系统表不同， `pg_shdepend` 是在集群里面所有的数据库之间共享的： 每个数据库集群只有一个 `pg_shdepend` ， 而不是每个数据库一个。

Table 47-40. `pg_shdepend` 字段

名字	类型	引用	描述
`dbid`	`oid`	`pg_database` .oid	依赖对象所在的数据库的 OID， 如果是共享对象， 则为零
`classid`	`oid`	`pg_class` .oid	依赖对象所在的系统表的 OID
`objid`	`oid`	任意 oid 属性	指定的依赖对象的 OID
`objsubid`	`int4`	对于一个表字段， 这是字段号 （ `objid` 和 `classid` 参考表本身） 。 对于所有其他对象类型， 这个字段为零。	
`refclassid`	`oid`	`pg_class` .oid	被引用对象所在的系统表的 OID(必须是一个共享表)
`refobjid`	`oid`	任意 oid 属性	指定的被引用对象的 OID
`deptype`	`char`	一段代码， 定义了这个依赖性关系的特定语义 ；参阅下文。	

在任何情况下， 一条 `pg_shdepend` 记录就表明这个被引用的对象不能在未删除依赖对象的前提下删除。 不过， `deptype` 同时还标出了几种不同的子风格 ：

`SHARED_DEPENDENCY_OWNER` (`o`)

被引用的对象(必须是一个角色)是依赖对象的所有者。

`SHARED_DEPENDENCY_ACL` (`a`)

被引用的对象(必须是一个角色)在依赖对象的 ACL(访问控制列表，也就是权限列表)里提到。`SHARED_DEPENDENCY_ACL` 不会在对象的所有者头上添加的，因为所有者会有一个 `SHARED_DEPENDENCY_OWNER` 记录。

`SHARED_DEPENDENCY_PIN` (`p`)

没有依赖对象；这类记录标识系统自身依赖于该被依赖对象，因此这样的对象绝对不能被删除。 这种类型的记录只是由 `initdb` 创建。这样的依赖对象的字段都是零。

其它依赖性的风格可能在将来会出现。请注意，目前的定义只是支持把角色当作被引用对象。

47.41. `pg_shdescription`

`pg_shdescription` 为共享数据库对象存储可选的注释。 可以使用COMMENT命令操作注释的内容，使用 psql的 `\d` 命令查看注释内容。

`pg_description` 提供了类似的功能， 它记录了单个数据库中对象的注释。

不同于大多数系统表， `pg_shdescription` 是在集群里面所有的数据库之间共享的： 每个数据库集群只有一个 `pg_shdescription` ， 而不是每个数据库一个。

Table 47-41. `pg_shdescription` 字段

名字	类型	引用	描述
`objoid`	`oid`	任意 oid 属性	这条描述所描述的对象的OID
`classoid`	`oid`	`pg_class` .oid	这个对象出现的系统表的OID
`description`	`text`	作为对该对象的描述的任意文本	

47.42. `pg_shseclabel`

`pg_shseclabel` 表存储在共享数据库对象上的安全标签。 安全标签可以用SECURITY LABEL命令操作。 查看安全标签的简单点的方法，请参阅Section 47.65。

`pg_seclabel` 表的作用类似， 只是它是用于在单个数据库内部的对象的安全标签的。

不同于大多数的系统表， `pg_shseclabel` 在一个集群中的所有数据库中共享： 每个数据库集群只有一个 `pg_shseclabel` ， 而不是每个数据库一个。

Table 47-42. `pg_shseclabel` 字段

名字	类型	引用	描述
`objoid`	`oid`	任意OID属性	这个安全标签所属的对象的OID
`classoid`	`oid`	`pg_class` .oid	出现这个对象的系统表的OID
`provider`	`text`	与这个标签相关的标签提供程序。	
`label`	`text`	应用于这个对象的安全标签。	

47.43. `pg_statistic`

`pg_statistic` 表存储有关该数据库内容的统计数据。 记录是由ANALYZE创建的，并且随后被查询规划器使用。 请注意所有统计信息天生都是近似的数值，即使假设它是最新的也如此。

通常这里对于每个被分析了的表字段有一条 `stainherit` = `false` 的记录。 如果该表有继承的子代，那么也会创建一条带有 `stainherit` = `true` 的记录。 这个行表示字段在继承树上的状态，也就是，对于这个数据的状态，你应该看到 `SELECT` `_column_` FROM `_table_` *，而 `stainherit` = `false` 行的结果为 `SELECT` `_column_` FROM ONLY `_table_` 。

`pg_statistic` 还存储有关索引表达式数值的统计数据。 这些是把他们当作实际的数据字段来描述的；特别是， `starelid` 引用索引。 不过，普通的非表达式索引字段没有记录，因为会和下层的表字段记录冗余。 当前，索引表达式的记录总是有 `stainherit` = `false` 。

因为不同类型的统计信息适用于不同类型的数据， `pg_statistic` 被设计成不太在意自己存储的是什么类型的统计。只有极为常用的统计信息(比如 NULL 的含量) 才在 `pg_statistic` 里给予专用的字段。其它所有东西都存储在 "槽位"中，而槽位是一组相关的字段，它们的内容用槽位中的一个字段的代码号码表示。 更详细的信息请参阅 `src/include/catalog/pg_statistic.h` 。

`pg_statistic` 不应该是公众可读的，因为即使是表内容的统计信息也应该认为是敏感的。 例子：薪水字段的最大最小值肯定是相当让人感兴趣的。 `pg_stats` 是一个在 `pg_statistic` 上的全局可读的视图，它只显示那些表对于当前用户可读的信息。

Table 47-43. `pg_statistic` 字段

名字	类型	引用	描述
`starelid`	`oid`	`pg_class` .oid	所描述的字段所属的表或者索引
`staattnum`	`int2`	`pg_attribute` .attnum	所描述的字段的个数
`stainherit`	`bool`	如果为真，那么统计数据包含继承子字段，不只是指定关系中的值。	
`stanullfrac`	`float4`	该字段中为 NULL 的记录的比率	
`stawidth`	`int4`	非 NULL 记录的平均存储宽度，以字节计	
`stadistinct`	`float4`	字段里唯一的非 NULL 数据值的数目。一个大于零的数值是独立数值的实际数目。 一个小于零的数值是表中行数的乘数的负数(比如，一个字段的数值平均出现概率为两次， 那么可以表示为 `stadistinct` = -0.5)。零值表示独立数值的数目未知。	
stakind``_N_	`int2`	一个编码，表示这种类型的统计存储在 `pg_statistic` 行的第 `_N_` 个"槽位"。	
staop``_N_	`oid`	`pg_operator` .oid	一个用于生成这些存储在第 `_N_` 个"槽位"的统计信息的操作符。 比如，一个柱面图槽位会显示 `<` 操作符，该操作符定义了该数据的排序顺序。
stanumbers``_N_	`float4[]`	第 `_N_` 个"槽位"的相关类型的数值统计， 如果该槽位和数值没有关系，那么就是 NULL 。	
stavalues``_N_	`anyarray`	第 `_N_` 个"槽位"相关类型的字段数据值， 如果该槽位类型不存储任何数据值那么就是 NULL 。 每个数组的元素值实际上都是指定字段的数据类型，或相关类型如一个数组的元素类型， 因此，除了把这些字段的类型定义成 `anyarray` 之外，没有更好的办法。	

47.44. `pg_tablespace`

`pg_tablespace` 存储有关可用的表空间的信息。 表可以放置在特定的表空间里，以帮助管理磁盘布局。

与大多数系统表不同， `pg_tablespace` 在一个集群中的所有数据库之间共享： 每个集群只有一份 `pg_tablespace` 的拷贝，而不是每个数据库一个。

Table 47-44. `pg_tablespace` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确指定)	
`spcname`	`name`	表空间名字	
`spcowner`	`oid`	`pg_authid`.oid	表空间的所有者，通常是创建它的人
`spcacl`	`aclitem[]`	访问权限；参阅GRANT和REVOKE获取细节。	
`spcoptions`	`text[]`	表空间级选项，使用"keyword=value"格式的字符串	

47.45. `pg_trigger`

`pg_trigger` 存储表和视图上面的触发器。 参阅CREATE TRIGGER获取更多信息。

Table 47-45. `pg_trigger` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`tgrelid`	`oid`	`pg_class` .oid	这个触发器所在的表
`tgname`	`name`	触发器名称(在同一表的所有触发器中必须唯一)	
`tgfoid`	`oid`	`pg_proc` .oid	要调用的函数
`tgtype`	`int2`	标识触发器条件的位掩码	
`tgenabled`	`char`	在session_replication_role模块里控制触发器的触发。 `O` = 触发器在"origin"和"local"模块里触发， `D` = 禁用触发器， `R` = 触发器在"replica"模块里触发， `A` = 触发器总是触发。	
`tgisinternal`	`bool`	如果触发器是内部产生的则为真(通常，强制约束由 `tgconstraint` 指定)	
`tgconstrrelid`	`oid`	`pg_class` .oid	一个参照完整性约束引用的表
`tgconstrindid`	`oid`	`pg_class` .oid	索引支持唯一、主键或参照完整性约束
`tgconstraint`	`oid`	`pg_constraint` .oid	`pg_constraint` 条目与触发器相关，如果有
`tgdeferrable`	`bool`	如果约束触发器可推迟则为真	
`tginitdeferred`	`bool`	如果约束触发器是初始可推迟则为真	
`tgnargs`	`int2`	传递给触发器函数的参数字符串个数	
			如果触发器是指

tgattr	int2vector	pg_attribute .attnum	定字段的则为字段号；否则为空数组
tgargs	bytea	传递给触发器的参数字符串，每个都是用 NULL 结尾	
tgqual	pg_node_tree	触发器的 WHEN 条件的表达式树(用 nodeToString() 表示)，如果没有则为空	

当前，指定字段的触发只支持 UPDATE 事件，因此 tgattr 只和事件类型相关。 tgtype 可能包含一些其他事件类型， 但是假定那些是在表范围的，不管 tgattr 里有什么。

> **Note:** 当 tgconstraint 非零时， tgconstrrelid , tgconstrindid , tgdeferrable , 和 tginitdeferred 与引用的 pg_constraint 条目在很大程度上是冗余的。然而，不可延缓的触发器与可延缓的约束关联是可能的： 外键约束可以有一些可延缓和不可延缓的触发器。
>
> **Note:** 如果一个关系有任何触发器在这个表里，则 pg_class.relhastriggers 必须为真。

47.46. `pg_ts_config`

`pg_ts_config` 表包含表示文本搜索配置的记录。 一个配置指定一个特定的文本搜索解析器和一个为了每个解析器的输出类型使用的字典的列表。 解析器在 `pg_ts_config` 记录中显示，但是字典映射的标记是由 `pg_ts_config_map` 里面的辅助记录定义的。

PostgreSQL的文本搜索功能在Chapter 12里面描述。

Table 47-46. `pg_ts_config` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`cfgname`	`name`	文本搜索配置名	
`cfgnamespace`	`oid`	`pg_namespace`.oid	包含这个配置的名字空间的OID
`cfgowner`	`oid`	`pg_authid` .oid	配置的所有者
`cfgparser`	`oid`	`pg_ts_parser`.oid	这个配置的文本搜索解析器的OID

47.47. pg_ts_config_map

pg_ts_config_map 表包含为每个文本搜索配置的解析器的每个输出符号类型，显示哪个文本搜索字典应该被咨询、以什么顺序搜索的记录。

PostgreSQL的文本搜索功能在Chapter 12里描述。

Table 47-47. pg_ts_config_map 字段

名字	类型	引用	描述
mapcfg	oid	pg_ts_config.oid	拥有这个映射记录的 pg_ts_config 记录的OID
maptokentype	integer		由配置的解析器发出的一个符号类型
mapseqno	integer		以什么顺序咨询这个记录(低 mapseqno 为先)
mapdict	oid	pg_ts_dict.oid	要咨询的文本搜索字典的OID

47.48. `pg_ts_dict`

`pg_ts_dict`表包含定义文本搜索字典的记录。 字典取决于文本搜索模板，该模板声明所有需要的实现函数； 字典本身提供模板支持的用户可设置的参数的值。这种分工允许字典通过非权限用户创建。 参数由文本字符串`dictinitoption`指定，参数的格式和意义取决于模板。

PostgreSQL的文本搜索功能在Chapter 12里描述。

Table 47-48. `pg_ts_dict` 字段

名子	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性; 必须明确选择)	
`dictname`	`name`	文本搜索字典名	
`dictnamespace`	`oid`	`pg_namespace`.oid	包含这个字段的名字空间的OID
`dictowner`	`oid`	`pg_authid`.oid	字典的所有者
`dicttemplate`	`oid`	`pg_ts_template`.oid	这个字典的文本搜索模板的OID
`dictinitoption`	`text`	该模板的初始化选项字符串	

47.49. `pg_ts_parser`

`pg_ts_parser` 表包含定义文本解析器的记录。 解析器负责分裂输入文本为词位，并且为每个词位分配标记类型。 因为解析器必须通过C语言级别的函数实现，所以新解析器的创建局限于数据库超级用户。

PostgreSQL的文本搜索功能在Chapter 12里描述。

Table 47-49. `pg_ts_parser` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性;必须明确选择)	
`prsname`	`name`	文本搜索解析器名	
`prsnamespace`	`oid`	`pg_namespace`.oid	包含这个解析器的名字空间的OID
`prsstart`	`regproc`	`pg_proc`.oid	解析器的启动函数的OID
`prstoken`	`regproc`	`pg_proc`.oid	解析器的下一个标记函数的OID
`prsend`	`regproc`	`pg_proc`.oid	解析器的关闭函数的OID
`prsheadline`	`regproc`	`pg_proc`.oid	解析器的标题函数的OID
`prslextype`	`regproc`	`pg_proc`.oid	解析器的lextype函数的OID

47.50. `pg_ts_template`

`pg_ts_template`表包含定义文本搜索模板的记录。 模板是文本搜索字典的类的实现框架。因为模板必须通过C语言级别的函数实现， 索引新模板的创建局限于数据库超级用户。

PostgreSQL的文本搜索功能在Chapter 12里描述。

Table 47-50. `pg_ts_template` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性；必须明确选择)	
`tmplname`	`name`	文本搜索模板名	
`tmplnamespace`	`oid`	`pg_namespace`.oid	包含这个模板的名字空间的OID
`tmplinit`	`regproc`	`pg_proc`.oid	模板的初始化函数的OID
`tmpllexize`	`regproc`	`pg_proc`.oid	模板的lexize函数的OID

47.51. `pg_type`

`pg_type` 存储有关数据类型的信息。基本类型和枚举类型(标量类型)是用 CREATE TYPE创建的，域是使用CREATE DOMAIN创建的。 同时还为数据库中每个表自动创建一个复合类型，以表示该表的行结构。还可以用 `CREATE TYPE AS` 创建复合类型。

Table 47-51. `pg_type` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性；必须明确选择)	
`typname`	`name`	数据类型名	
`typnamespace`	`oid`	`pg_namespace`.oid	包含这个类型的名字空间的OID
`typowner`	`oid`	`pg_authid`.oid	该类型的所有者
`typlen`	`int2`	对于定长类型是该类型内部表现形式的字节数目。对于变长类型是负数。 -1 表示一种"变长"类型(有长度字属性的数据)， -2 表示这是一个 NULL 结尾的 C 字符串。	
`typbyval`	`bool`	判断内部过程传递这个类型的数值时是通过传值还是传引用。 如果该类型不是 1, 2, 4, 8 字节长将只能按引用传递，因此 typbyval 最好是假。 即使可以传值，typbyval 也可以为假。	
`typtype`	`char`	对于基础类型是 `b`， 对于复合类型是 `c` (比如，一个表的行类型)。对于域类型是 `d`， 对于枚举类型是 `e`， 对于伪类型是 `p`， 对于范围类型是 `r`。 又见 `typrelid` 和 `typbasetype`。	
`typcategory`	`char`	`typcategory` 是数据类型的任意分类， 该数据类型被触发器用来决定哪种隐式转换应该是"首选"。 参阅Table 47-52。	
`typispreferred`	`bool`	如果类型在它的 `typcategory` 里是首选转换目标则为真。	
		如果定义了类型则为真，如果	

typisdefined	bool	是一种尚未定义的类型的占位符则为假。如果为假，那么除了该类型名称，名字空间，和 OID 之外没有可靠的信息。	
`typdelim`	`char`	当分析数组输入时，分隔两个此类型数值的字符。请注意该分隔符是与数组元素数据类型相关联的，而不是和数组数据类型关联。	
typrelid	oid	pg_class .oid	如果是复合类型(见 typtype)那么这个字段指向 pg_class 中定义该表的行。对于自由存在的复合类型， pg_class 记录并不表示一个表，但是总需要它来查找该类型连接的 pg_attribute 记录。对于非复合类型为零。
`typelem`	`oid`	`pg_type` .oid	如果不为 0 ，那么它标识 `pg_type` 里面的另外一行。当前类型可以像一个数组产生类型为 `typelem` 的值一样当做下标。一个"真正的"数组类型是变长的(`typlen` = -1)，但是一些定长的(`typlen` > 0)类型也拥有非零的 `typelem` (比如 `name` 和 `point`)。如果一个定长类型拥有一个 `typelem` ，那么他的内部形式必须是 `typelem` 数据类型的某个数目的个数值，不能有其它数据。变长数组类型有一个该数组子过程定义的头(文件)。
typarray	oid	pg_type .oid	如果 typarray 非零，那么它在 pg_type 里定义另外一行，该行是将这个类型作为元素的"真正的"数组类型。

`typinput`	`regproc`	`pg_proc` .oid	输入转换函数(文本格式)
`typoutput`	`regproc`	`pg_proc` .oid	输出转换函数(文本格式)
`typreceive`	`regproc`	`pg_proc` .oid	输入转换函数(二进制格式)，如果没有则为 0
`typsend`	`regproc`	`pg_proc` .oid	输出转换函数(二进制格式)，如果没有则为 0
`typmodin`	`regproc`	`pg_proc` .oid	类型修饰符输入函数，如果类型不支持修饰符则为0
`typmodout`	`regproc`	`pg_proc` .oid	类型修饰符输出函数，如果使用标准格式则为0
`typanalyze`	`regproc`	`pg_proc` .oid	自定义的 `ANALYZE` 函数，如果使用标准函数，则为 0
`typalign`	`char`	当存储此类型的数值时要求的对齐性质。它应用于磁盘存储以及该值在PostgreSQL 内部的大多数形式。如果数值是连续存放的，比如在磁盘上以完全的裸数据的形式存放时， 那么先在此类型的数据前填充空白，这样它就可以按照要求的界限存储。对齐引用是该序列中第一个数据的开头。	

可能的值有：

- `c` = `char` 对齐，也就是不需要对齐。
- `s` = `short` 对齐(在大多数机器上是 2 字节)
- `i` = `int` 对齐(在大多数机器上是 4 字节)
- `d` = `double` 对齐(在大多数机器上是 8 字节，但不一定是全部)

> **Note:** 对于在系统表里使用的类型，在 `pg_type` 里定义的尺寸和对齐必须和编译器在一个表示表的一行的结构里的布局一样。 || `typstorage` | `char` | 告诉一个变长类型(那些有 `typlen` = -1的) 说该类型是否准备好应付非常规值，以及对这种类型的属性的缺省策略是什么。可能的值有

- `p` :数值总是以简单方式存储

- `e` : 数值可以存储在一个"次要"关系中(如果该关系有这么一个， 参阅 `pg_class.reltoastrelid`)
- `m` : 数值可以以内联的压缩方式存储
- `x` : 数值可以以内联的压缩方式或者在"次要"表里存储。

请注意 `m` 域也可以移到从属表里存储，但只是最后的解决方法 (`e` 和 `x` 域先移走)。 || `typnotnull` | `bool` | 代表在某类型上的一个 NOTNULL 约束。目前只用于域。 || `typbasetype` | `oid` | `pg_type` .oid | 如果这是一个域(参阅 `typtype`)， 那么标识作为这个类型的基础的类型。如果不是域则为零。 || `typtypmod` | `int4` | 域使用 `typtypmod` 记录要作用到它们的基础类型上的 `typmod` (如果基础类型不使用 `typmod` 则为 -1)。如果这种类型不是域，那么为 -1 。 || `typndims` | `int4` | 如果是一个域数组，那么 `typndims` 是数组维数的数值(也就是说， `typbasetype` 是一个数组类型）。非域非数组类型为零。 || `typcollation` | `oid` | `pg_collation` .oid | 指定类型的排序规则。如果类型不支持排序，则为0。一个支持排序的基础类型将有 `DEFAULT_COLLATION_OID` 。一个可排序类型的域可以有一些其他排序OID，如果为该域指定了一个的话。 || `typdefaultbin` | `pg_node_tree` | 如果为非 NULL ，那么它是该类型缺省表达式的 `nodeToString()` 表现形式。 目前这个字段只用于域。 || `typdefault` | `text` | 如果某类型没有相关缺省值，那么 `typdefault` 是 NULL 。如果 `typdefaultbin` 不是 NULL ，那么 `typdefault` 必须包含一个 `typdefaultbin` 代表的缺省表达式的人类可读的版本。如果 `typdefaultbin` 为 NULL 但 `typdefault` 不是，那么 `typdefault` 是该类型缺省值的外部表现形式， 可以把它交给该类型的输入转换器生成一个常量。 || `typacl` | `aclitem[]` | 访问权限；参阅GRANT和REVOKE获取细节。 |

Table 47-52列出了系统定义的 `typcategory` 的值。 任何未来添加到这个列表的也是大写的ASCII字母。所有其他ASCII字符为用户定义的范畴保留。

Table 47-52. `typcategory` 代码

代码	种类
`A`	数组类型
`B`	布尔类型
`C`	复合类型
`D`	日期/时间类型
`E`	枚举类型
`G`	几何类型
`I`	网络地址类型
`N`	数值类型
`P`	伪类型
`R`	范围类型
`S`	字符串类型
`T`	时间间隔类型
`U`	用户定义类型
`V`	位串类型
`X`	`未知` 类型

47.52. `pg_user_mapping`

`pg_user_mapping`表存储从本地用户到远程的映射。 普通用户访问这个表是受到限制的，使用视图访问 `pg_user_mappings` 。

Table 47-53. `pg_user_mapping` 字段

名字	类型	引用	描述
`oid`	`oid`	行标识符(隐藏属性；必须明确选择)	
`umuser`	`oid`	`pg_authid` .oid	被映射的本地用户的OID，如果用户映射是公共的则为0
`umserver`	`oid`	`pg_foreign_server` .oid	包含这个映射的外部服务器的OID
`umoptions`	`text[]`	用户映射指定选项，使用"keyword=value"格式的字符串	

47.53. 系统视图

除了系统表之外，PostgreSQL还提供了一系列内置的视图。 系统视图提供了查询系统表的一些便利的访问方法。 其它一些视图提供了访问内部服务器状态的方法。

信息模式(Chapter 34)提供了另外一套视图， 它的功能覆盖了系统视图的功能。因为信息模式是 SQL 标准， 而这里描述的视图是PostgreSQL特有的， 所以最好用信息模式来获取自己需要的所有信息。

Table 47-54列出了这里描述的所有系统视图。下面是每个视图更详细的信息。 有些视图提供了对统计收集器的结果的访问；他们在Table 27-1里列出。

除了特别声明的，这里描述的所有视图都是只读的。

Table 47-54. 系统视图

视图名	用途
pg_available_extensions	可用的扩展
pg_available_extension_versions	可用扩展的版本
pg_cursors	打开的游标
pg_group	数据库用户的组
pg_indexes	索引
pg_locks	当前持有的锁
pg_matviews	物化视图
pg_prepared_statements	预备语句
pg_prepared_xacts	预备事务
pg_roles	数据库角色
pg_rules	规则
pg_seclabels	安全标签
pg_settings	参数设置
pg_shadow	数据库用户
pg_stats	规划器统计
pg_tables	表
pg_timezone_abbrevs	时区缩写
pg_timezone_names	时区名
pg_user	数据库用户
pg_user_mappings	用户映射
pg_views	视图

47.54. `pg_available_extensions`

`pg_available_extensions` 视图列出了可用于安装的扩展。 又见 `pg_extension` 表， 显示了当前安装了的扩展。

Table 47-55. `pg_available_extensions` 字段

名字	类型	描述
`name`	`name`	扩展名
`default_version`	`text`	缺省版本的名字，如果没有指定则为 `NULL`
`installed_version`	`text`	扩展当前安装版本，如果没有安装任何版本则为 `NULL`
`comment`	`text`	扩展的控制文件中的评论字符串

`pg_available_extensions` 视图是只读的。

47.55. `pg_available_extension_versions`

`pg_available_extension_versions` 视图列出了可用于安装的指定扩展版本。 又见 `pg_extension` 表， 显示了当前安装了的扩展。

Table 47-56. `pg_available_extension_versions` 字段

名字	类型	描述
`name`	`name`	扩展名
`version`	`text`	版本名
`installed`	`bool`	如果这个扩展的这个版本是当前已经安装了的则为真
`superuser`	`bool`	如果只允许超级用户安装这个扩展则为真
`relocatable`	`bool`	如果扩展可以重新加载到另一个模式则为真
`schema`	`name`	扩展必须安装到的模式名，如果部分或全部可重新定位则为 `NULL`
`requires`	`name[]`	先决条件扩展的名字，如果没有则为 `NULL`
`comment`	`text`	扩展的控制文件中的评论字符串

`pg_available_extension_versions` 是只读的。

47.56. `pg_cursors`

`pg_cursors` 列出了当前可用的游标。游标可以用几种不同的方法定义：

- 通过DECLARE语句
- 在前/后端协议中通过 Bind 信息，具体在Section 48.2.3中描述。
- 通过服务器编程接口(SPI)，具体在Section 44.1中描述。

`pg_cursors` 显示上述所有方法创建的游标。 除非被声明为 `WITH HOLD`，游标仅存在于定义它们的事务的生命期中。 因此非持久游标仅能够在视图中存在到创建该游标的事务结束时为止。

> **Note:** 因为游标用于在PostgreSQL内部实现一些比如过程语言之类的组件。 因此`pg_cursors`可能包含并非由用户明确创建的游标。

Table 47-57. `pg_cursors` 字段

名字	类型	描述
`name`	`text`	游标名
`statement`	`text`	声明该游标的查询字符串
`is_holdable`	`boolean`	如果该游标是持久的(也就是在声明该游标的事务结束后仍然可以访问该游标)则为 `true` ； 否则为 `false`
`is_binary`	`boolean`	如果该游标被声明为 `BINARY` 则为 `true` ； 否则为 `false`
`is_scrollable`	`boolean`	如果该游标可以滚动(也就是允许以不连续的方式检索)则为 `true` ； 否则为 `false`
`creation_time`	`timestamptz`	声明该游标的时间戳

`pg_cursors` 视图是只读的。

47.57. `pg_group`

`pg_group` 的存在是为了向下兼容：它模拟一个存在于PostgreSQL 版本 8.1 之前的系统表。它显示所有标记为不是 `rolcanlogin` 的角色的名字和成员， 这就是近似于用做组的那些角色的集合了。

Table 47-58. `pg_group` 字段

名字	类型	引用	描述
`groname`	`name`	`pg_authid`.rolname	组的名字
`grosysid`	`oid`	`pg_authid` .oid	组的 ID
`grolist`	`oid[]`	`pg_authid`.oid	一个数组，包含这个组里面所有角色的 ID

47.58. pg_indexes

pg_indexes 提供对数据库中每个索引的有用信息的访问。

Table 47-59. pg_indexes 字段

名字	类型	引用	描述
schemaname	name	pg_namespace.nspname	包含表和索引的模式的名字
tablename	name	pg_class.relname	此索引所服务的表的名字
indexname	name	pg_class.relname	索引的名字
tablespace	name	pg_tablespace.spcname	包含索引的表空间名字(如果是数据库缺省，则为 NULL)
indexdef	text	索引定义(一个重建的 CREATE INDEX 命令)	

47.59. `pg_locks`

`pg_locks` 提供有关在数据库服务器中由打开的事务持有的锁的信息。 参阅Chapter 13获取有关锁的更多的讨论。

`pg_locks` 对每个活跃的可锁定对象、请求的锁模式、 以及相关的事务保存一行。因此，如果多个事务持有或者等待对同一个对象的锁， 那么同一个可锁定的对象可能出现多次。不过，一个目前没有锁在其上的对象将肯定不会出现。

有好几种不同的可锁定对象：一个关系(也就是一个表)、关系中独立页面、 关系中独立的行、一个事务 ID（虚拟和永久ID）、以及一般的数据库对象 (用类别 OID 和对象 OID 标识，表示方法和 `pg_description` 或 `pg_depend` 一样)还有，扩展一个关系的权限也是用一种独立的可锁定对象表示的。 另外，"advisory"锁可以用在有用户定义的含义的数据上。

Table 47-60. `pg_locks` 字段

名字	类型	引用	描述
`locktype`	`text`	可锁定对象的类型： `relation` , `extend` , `page` , `tuple` , `transactionid` , `virtualxid` , `object` , `userlock` , 或 `advisory`	
`database`	`oid`	`pg_database` .oid	目标所在的数据库的 OID， 如果目标是共享对象，那么就是零， 如果目标是一个事务 ID， 就是 null 。
`relation`	`oid`	`pg_class` .oid	关系的 OID， 如果目标不是关系，也不是关系的一部分， 则为 null
`page`	`integer`	关系内部的页面编号，如果目标不是行页不是关系页，则为null	
`tuple`	`smallint`	页面里面的行编号，如果目标不是行，则为 null	
`virtualxid`	`text`	事务的虚拟 ID， 如果目标不是虚拟事务 ID， 就是 null	
`transactionid`	`xid`	事务的 ID， 如果目标不是事务 ID， 就是 null	

`classid`	`oid`	`pg_class` .oid	包含该目标的系统表的 OID， 如果目标不是普通数据库对象，则为 null
`objid`	`oid`	任意OID属性	目标在其系统表内的 OID， 如果目标不是普通数据库对象，则为 null
`objsubid`	`smallint`	字段编号 (`classid` 和 `objid` 指向表自身)。 如果目标是其它普通数据库对象，这个字段是零。如果这个目标不是普通数据库对象，则为 null	
`virtualtransaction`	`text`	持有此锁或者在等待此锁的事务的虚拟 ID	
`pid`	`integer`	持有或者等待这个锁的服务器进程的进程 ID 。如果锁是被一个预备事务持有的，那么为 null	
`mode`	`text`	这个进程持有的或者是期望的锁模式(参阅Section 13.3.1 和 Section 13.2.3)	
`granted`	`boolean`	如果持有锁，为真，如果等待锁，为假	
`fastpath`	`boolean`	如果锁通过快速路径获得为真，如果通过主锁表获得为假	

`granted` 为真时表明指定事务持有一个锁。为假则表明该事务当前等待使用这个锁， 这就暗示着某个其它的事务正在同样的可锁定对象上持有冲突的锁模式。等待的会话将一直睡眠，直到另外一个锁释放(或者侦测到一个死锁条件)。一个事务一次最多等待一个锁。

每个事务都在它持续的时间里在他自己的虚拟事务 ID 上持有一个排他锁。如果给事务赋予一个永久ID （通常只在事务改变数据库的状态时发生），那么它也在它的永久事务ID上持有一个排它锁，直到它结束。 如果一个事务认为它必须等待另外一个事务，它会以企图在另外一个事务 ID 上获取共享锁的方式实现之 （虚拟的或永久的ID取决于情景）。这个锁只有在另外一个事务终止并且释放它的锁的前提下才能成功。

尽管行是一种可以锁定的对象，但是有关行级别锁的信息是存储在磁盘上的，而不是在内存里， 因此，行级别的锁通常不会出现在这个视图里。如果一个事务在等待一个行级别的锁，那么它通常会在这个视图里以等待当前持有该行锁的永久事务 ID 的方式出现。

建议锁可以在由单独一个 `bigint` 值或两个integer值组成的键上获得。 一个 `bigint` 键的高/低位部分分别在 `classid` 和 `objid` 字段中显示，并且 `objsubid` 等于 1 。原先的 `bigint` 值可以通过 表达式 `(classid::bigint << 32) | objid::bigint` 重新组装。 integer 组成的键前半部分在 `classid` 字段中显示、后半部分在 `objid` 字段中显示，并且 `objsubid` 等于 2 。 键的实际含义取决于用户的定义。 建议锁是针对单个数据库的，因此 `database` 字段对于建议锁就显得很有意义了。

`pg_locks` 提供了一个数据库集群里的所有的锁的全局视图， 而不仅仅那些和当前数据库相关的。尽管它的 `relation` 字段可以和 `pg_class` . `oid` 连接起来以标识被锁住的关系， 但是这个方法目前只能对在当前数据库里的关系有用(那些 `database` 字段是当前数据库的 OID 或者零的数据库)。

`pid` 字段可以可以和 `pg_stat_activity` 视图的 `pid` 字段连接起来获取持有或者等待持有每个锁的会话的更多信息。 同样，如果你使用预备事务，可以把 `transaction` 字段和 `pg_prepared_xacts` 视图的 `transaction` 字段连接起来获取持有锁的那个预备事务的更多信息。 一个预备事务不能等待任何锁，但是在运行的时候，它继续持有它已经请求到的锁。

`pg_locks` 显示独立系统的普通锁管理器和谓词锁管理器的数据； 另外，普通锁管理器细分它的锁为普通和*fast-path*锁。这个数据不保证是完全一致的。 当请求视图时，fast-path锁的数据（ `fastpath` = `true` ） 从每个后端一次收集，在整个锁管器中没有冻结状态，所以当收集信息时，可以获取锁或释放锁。 不过，要注意的是，这些锁不和任意其他当前锁发生冲突。 在所有后端已经查询fast-path锁之后， 剩余的普通锁管理器作为一个单元，并且一个所有剩余锁的一致的快照作为原子动作收集。 在解锁普通锁管理器之后，谓词锁管理器同样的锁住，并且所有谓词锁作为一个原子动作收集。 因此，除了fast-path锁，每个锁管理器将给出一致的结果集，但是因为我们不同时锁住锁管理器， 锁可能在我们访问普通锁管理器之后、访问谓词锁管理器之前获取或释放。

如果这个视图访问的非常频繁，那么锁住普通和/或谓词锁管理器可能会对数据库性能有些影响。 锁只持有从锁管理器获取数据所需要的最小的时间，但是这并不能完全消除性能影响的可能性。

47.60. `pg_matviews`

`pg_matviews` 视图提供了访问关于每个物化视图在数据库中的有用信息的接口。

Table 47-61. `pg_matviews` 字段

名字	类型	引用	描述
`schemaname`	`name`	`pg_namespace`.nspname	包含物化视图的模式名
`matviewname`	`name`	`pg_class`.relname	物化视图的名字
`matviewowner`	`name`	`pg_authid`.rolname	物化视图的所有者的名字
`tablespace`	`name`	`pg_tablespace`.spcname	包含物化视图的名字空间的名字（如果对于数据库是缺省的则为null）
`hasindexes`	`boolean`	如果物化视图有（或最近有）任何索引则为真	
`ispopulated`	`boolean`	如果物化视图当前填充了则为真	
`definition`	`text`	物化视图定义（一个重新构造的 `SELECT` 查询）	

47.61. `pg_prepared_statements`

`pg_prepared_statements` 显示所有当前会话中可用的预备语句。 参见PREPARE获取关于预备语句的更多信息。

每个预备语句在 `pg_prepared_statements` 中都有对应的一条记录。 当一条新的预备语句创建后该视图中就会新增一条记录，同样， 当一条预备语句被释放后(比如通过DEALLOCATE命令)，相应的记录也会被删除。

Table 47-62. `pg_prepared_statements` 字段

名字	类型	描述
`name`	`text`	预备语句的标识符
`statement`	`text`	创建该预备语句的查询字符串。对于从 SQL 创建的预备语句而言是客户端提交的 PREPARE 语句。 对于通过前/后端协议创建的预备语句而言是预备语句自身的文本。
`prepare_time`	`timestamptz`	创建该预备语句的时间戳
`parameter_types`	`regtype[]`	该预备语句期望的参数类型，以 `regtype` 类型的数组格式出现。 与该数组元素相对应的 OID 可以通过把 `regtype` 值转换为 `oid` 值得到。
`from_sql`	`boolean`	如果该预备语句是通过 PREPARE 语句创建的则为 `true` ； 如果是通过前/后端协议创建的则为 `false`

`pg_prepared_statements` 视图是只读的。

47.62. `pg_prepared_xacts`

`pg_prepared_xacts` 显示那些当前准备好进行两阶段提交的事务的信息 (参阅PREPARE TRANSACTION获取细节)。

`pg_prepared_xacts` 为每个预备事务包含一行。如果事务提交或者回滚， 则删除该条记录。

Table 47-63. `pg_prepared_xacts` 字段

名字	类型	引用	描述
`transaction`	`xid`	预备事务的数字事务标识	
`gid`	`text`	赋予该事务的全局事务标识	
`prepared`	`timestamp with time zone`	事务准备好提交的时间	
`owner`	`name`	`pg_authid` .rolname	执行该事务的用户的名字
`database`	`name`	`pg_database` .datname	执行该事务所在的数据库名

在访问 `pg_prepared_xacts` 视图的时候，内部事务管理器数据结构被暂时锁住， 并且为显示视图制作了一份拷贝。这样就保证了视图生成一个一致的结果集， 而不会阻塞正常的操作太长时间。当然，即便这么做，如果过于频繁地访问这个视图， 肯定也会对数据库性能造成一定的影响。

47.63. `pg_roles`

`pg_roles` 提供访问数据库角色有关信息的接口。 它只是一个 `pg_authid` 表的公开可读部分的视图，把口令字段用空白填充了。

该视图明确的显示了底层表的 OID 字段，可以用于与其它表连接。

Table 47-64. `pg_roles` 字段

名字	类型	引用	描述
`rolname`	`name`	角色名	
`rolsuper`	`bool`	有超级用户权限的角色	
`rolinherit`	`bool`	自动继承属主角色权限的角色	
`rolcreaterole`	`bool`	可以创建更多角色的角色	
`rolcreatedb`	`bool`	可以创建数据库的角色	
`rolcatupdate`	`bool`	可以直接更新系统表的角色。除非这个字段为真，否则超级用户也不能干这个事情。	
`rolcanlogin`	`bool`	可以登录的角色，也就是说，这个角色可以给予初始化会话认证的标识符。	
`rolreplication`	`bool`	复制的角色。也就是说，这个角色可以初始化流复制(参阅Section 25.2.5) 和使用 `pg_start_backup` 和 `pg_stop_backup` 设置/重设系统备份模式。	
`rolconnlimit`	`int4`	对于可以登录的角色，这儿限制了该角色允许发起的最大并发连接数。 -1 表示无限制。	
`rolpassword`	`text`	不是口令(总是 `********`)	
`rolvaliduntil`	`timestamptz`	口令失效日期(只用于口令认证)；如果没有失效期，为 NULL	
`rolconfig`	`text[]`	运行时配置变量的用户指定的缺省	
`oid`	`oid`	`pg_authid`.oid	角色的ID

47.64. `pg_rules`

`pg_rules` 提供对查询重写规则的有用信息访问的接口。

Table 47-65. `pg_rules` 字段

名字	类型	引用	描述
`schemaname`	`name`	`pg_namespace`.nspname	包含表的模式的名字
`tablename`	`name`	`pg_class`.relname	规则作用的表的名字
`rulename`	`name`	`pg_rewrite`.rulename	规则的名字
`definition`	`text`	规则定义(一个重新构造的创建命令)	

`pg_rules` 视图排除了视图和物化视图的 `ON SELECT` 规则； 就是那些可以在 `pg_views` 和 `pg_matviews` 里看到的。

47.65. pg_seclabels

pg_seclabels 提供关于安全标签的信息。它是一个 pg_seclabel 表容易查询的版本。

Table 47-66. pg_seclabels 字段

名字	类型	引用	描述
objoid	oid	任意OID属性	这个安全标签指向的对象的OID
classoid	oid	pg_class .oid	这个对象出现的系统表的OID
objsubid	int4	对于一个在表字段上的安全标签，是字段编号（引用表本身的 objoid 和 classoid ）。对于所有其他对象类型，这个字段为零。	
objtype	text	这个标签出现的对象的类型，文本格式。	
objnamespace	oid	pg_namespace .oid	这个对象的名字空间的OID，如果适用；否则为NULL。
objname	text	这个标签适用的对象的名字，文本格式。	
provider	text	pg_seclabel .provider	与这个标签相关的标签提供者。
label	text	pg_seclabel .label	适用于这个对象的安全标签。

47.66. pg_settings

`pg_settings` 提供了对服务器运行时参数的访问。 它实际上是SHOW和SET命令的另外一个接口。 它还提供一些用 `SHOW` 不能直接获取的参数的访问，比如最大和最小值。

Table 47-67. `pg_settings` 字段

名字	类型	描述
`name`	`text`	运行时配置参数名
`setting`	`text`	参数的当前值
`unit`	`text`	参数的隐含单元
`category`	`text`	参数的逻辑组
`short_desc`	`text`	参数的一个简短的描述
`extra_desc`	`text`	有关参数的额外的，更详细的描述
`context`	`text`	设置这个参数的值要求的环境（见下文）
`vartype`	`text`	参数类型(`bool` ， `enum` ， `integer` ， `real` ， `string`)
`source`	`text`	当前参数值的来源
`min_val`	`text`	该参数允许的最小值(非数字值为 null)
`max_val`	`text`	这个参数允许的最大的数值(非数字值为 null)
`enumvals`	`text[]`	枚举参数允许的值（非枚举值为null）
`boot_val`	`text`	如果参数没有设置则为服务器启动时假设的参数值
`reset_val`	`text`	`RESET` 在当前会话中将重设的参数值
`sourcefile`	`text`	设置当前值的配置文件（从源码而不是配置文件设置值或当通过非超级用户检查时为null）； 当在配置文件中使用 `include` 指令时是有帮助的。
`sourceline`	`integer`	设置当前值的配置文件中的行编码 （从源码而不是配置文件设置值或当通过非超级用户检查时为null）

有几个 `context` 的可能值。以减少困难的改变设置的顺序，它们是：

`internal`

不能直接更改这些设置；它们反映了内部确定的值。其中的一些可以通过用不同的配置选项重建服务器， 或通过改变提供给 `initdb` 的选项来更改。

`postmaster`

这些选项只在服务器启动时使用，所以任何改变都需要重启服务器。这些设置的值通常存储在 `postgresql.conf` 文件中，或当服务器启动时传递给命令行。 当然，带有任何低 `context` 类型的设置也可以在服务器启动时设置。

`sighup`

这些设置可以在 `postgresql.conf` 中改变而不用重启服务器。 发送一个SIGHUP信号到主进程使其重读 `postgresql.conf` 并应用改变。主进程也将SIGHUP信号传递给它的子进程， 这样它们所有都使用新值。

`backend`

这些设置可以在 `postgresql.conf` 中改变而不用重启服务器； 它们也可以在连接需求包中为特定的会话设置（例如，通过libpq的 `PGOPTIONS` 环境变量）。不过，这些设置在会话启动后永远不会改变。 如果你在 `postgresql.conf` 中改变了它们，那么发送一个SIGHUP 信号到主进程使其重读 `postgresql.conf` 。新值将只影响随后加载的会话。

`superuser`

这些值可以在 `postgresql.conf` 中设置，或在一个会话中通过 `SET` 命令设置；但是只有超级用户可以通过 `SET` 改变它们。在 `postgresql.conf` 中改变它们将只在没有会话本地值是使用 `SET` 建立的时影响现有会话。

`user`

这些值可以在 `postgresql.conf` 中设置，或在一个会话中通过 `SET` 命令设置。允许任何用户改变他们的会话本地值。 `postgresql.conf` 中的改变将只在没有会话本地值是使用 `SET` 建立的时影响现有会话。

参阅Section 18.1获取更多关于改变这些参数的各种方式的信息。

不能对 `pg_settings` 视图进行插入或者删除，但是可以更新。 对 `pg_settings` 中的一行进行 `UPDATE` 等效于在该命名参数上执行SET命令。这个修改只影响当前会话使用的数值。 如果在一个最后退出的事务中发出了 `UPDATE` 命令，那么 `UPDATE` 命令的效果将在事务回滚之后消失。一旦包围它的事务提交，这个效果将固化，直到会话结束， 除非由其它的 `UPDATE` 或 `SET` 命令覆盖。

47.67. `pg_shadow`

`pg_shadow` 存在是为了向下兼容：它模拟了一个PostgreSQL 版本 8.1 之前的系统表。它显示了所有在 `pg_authid` 中标记了 `rolcanlogin` 的角色的属性。

这个系统表的名字来自于该表不能被公众可读，因为它包含口令。 `pg_user` 是一个在 `pg_shadow` 上公开可读的视图，只是把口令域填成了空白。

Table 47-68. `pg_shadow` 字段

名字	类型	引用	描述
`usename`	`name`	`pg_authid`.rolname	用户名
`usesysid`	`oid`	`pg_authid`.oid	用户的ID
`usecreatedb`	`bool`	用户可以创建数据库	
`usesuper`	`bool`	用户是超级用户	
`usecatupd`	`bool`	用户可以更新系统表。即使超级用户，如果这个字段不是真，也不能更新系统表。	
`userepl`	`bool`	用户可以初始化流复制和使系统处于或不处于备份模式。	
`passwd`	`text`	口令(可能是加密的)；如果没有则为null。参阅 `pg_authid` 获取加密的口令是如何存储的信息。	
`valuntil`	`abstime`	口令失效的时间(只用于口令认证)	
`useconfig`	`text[]`	运行时配置变量的会话缺省	

47.68. `pg_stats`

`pg_stats` 提供对存储在 `pg_statistic` 表里面的信息的访问。 这个视图允许只访问那些在 `pg_statistic` 里面对应用户有权限读取的表的数据行， 因此可以安全地允许公众访问这个视图。

`pg_stats` 也设计成把信息以一种更易读的方式出现的形式， 它比下层的系统表更容易阅读，代价就是如果在 `pg_statistic` 里定义了新的数据槽位，那么必须扩展它的视图定义。

Table 47-69. `pg_stats` 字段

名字	类型	引用	描述
`schemaname`	`name`	`pg_namespace`.nspname	包含此表的模式名字
`tablename`	`name`	`pg_class` .relname	表的名字
`attname`	`name`	`pg_attribute`.attname	这一行描述的字段的名字
`inherited`	`bool`	如果为真，那么这行包含继承的子字段，不只是指定表的值。	
`null_frac`	`real`	记录中字段为空的百分比	
`avg_width`	`integer`	字段记录以字节记的平均宽度	
		如果大于零，就是在字段中独立数值的估计数目。如果小于零， 就是独立数值的数目被行数除的负数。用负数形式是	

n_distinct	real	因为 ANALYZE 认为独立数值的数目是随着表增长而增长；正数的形式用于在字段看上去好像有固定的可能值数目的情况下。比如， -1 表示一个唯一字段，独立数值的个数和行数相同。
most_common_vals	anyarray	一个字段里最常用数值的列表。如果看上去没有啥数值比其它更常见，则为 null
most_common_freqs	real[]	一个最常用数值的频率的列表，也就是说，每个出现的次数除以行数。 如果 most_common_vals 是 null ，则为 null。
histogram_bounds	anyarray	一个数值的列表，它把字段的数值分成几组大致相同热门的组。 如果在 most_common_vals 里有数值，则在这个饼图的计算中省略。 如果字段数据类型没有 < 操作符或者 most_common_vals 列表代表了整个分布性，则这个字段为 null。
correlation	real	统计与字段值的物理行序和逻辑行序有关。它的范围从 -1 到 +1 。 在数值接近 -1 或者 +1 的时候，在字段上的索引扫描将被认为比它接近零的时候开销更少， 因为减少了对磁盘的随机访问。如果字段数据类型没有 < 操作符，那么这个字段为null。
most_common_elems	anyarray	经常在字段值中出现的非空元素值的列表。（标量类型为空。）
most_common_elem_freqs	real[]	最常见元素值的频率列表，也就是，至少包含一个给定值的实例的行的分数。 每个元素频率跟着两到三个附加的值；它们是在每个元素频率之前的最小和最大值， 还有可选择的null元素的频率。（当 most_common_elems 为null时，为 null）
elem_count_histogram	real[]	该字段中值的不同非空元素值的统计直方图，跟着不同非空元素的平均值。（标量类型为空。）

在数组里的元素的最大数目可以用 ALTER TABLE SET STATISTICS 命令一个一个字段地控制， 或者通过设置运行时参数default_statistics_target全局地设置。

47.69. `pg_tables`

`pg_tables` 提供了对有关数据库中每个表的有用信息地访问。

Table 47-70. `pg_tables` 字段

名字	类型	引用	描述
`schemaname`	`name`	`pg_namespace`.nspname	包含表的模式名字
`tablename`	`name`	`pg_class`.relname	表的名字
`tableowner`	`name`	`pg_authid`.rolname	表的所有者的名字
`tablespace`	`name`	`pg_tablespace`.spcname	包含表的表空间名字(如果是数据库缺省，则为 null)
`hasindexes`	`boolean`	`pg_class`.relhasindex	如果表拥有(或者最近拥有)任何索引，则为真
`hasrules`	`boolean`	`pg_class`.relhasrules	如果表有（或曾静有）规则，则为真
`hastriggers`	`boolean`	`pg_class`.relhastriggers	如果表有（或曾静有）触发器，则为真

47.70. `pg_timezone_abbrevs`

`pg_timezone_abbrevs` 提供了输入例程能够识别的所有时区缩写。 当运行时参数timezone_abbreviations发生改变的时候，该视图的内容也会发生改变。

Table 47-71. `pg_timezone_abbrevs` 字段

名字	类型	描述
`abbrev`	`text`	时区缩写
`utc_offset`	`interval`	相对于 UTC 的偏移量
`is_dst`	`boolean`	如果这是一个夏时制时区缩写则为真

47.71. `pg_timezone_names`

`pg_timezone_names` 显示了所有能够被 `SET TIMEZONE` 识别的时区名及其缩写、UTC 偏移量、是否夏时制。（学术上，PostgreSQL 使用UT1而不是UTC，因为它不处理闰秒。）不同于在 `pg_timezone_abbrevs` 中显示的缩写，许多这些名字都隐含着夏令时转换规则。因此，在跨越夏令时边界时相关信息会发生变化。 显示的信息给予当前的 `CURRENT_TIMESTAMP` 值进行计算。

Table 47-72. `pg_timezone_names` 字段

名字	类型	描述
`name`	`text`	时区名
`abbrev`	`text`	时区缩写
`utc_offset`	`interval`	相对于 UTC 的偏移量（正数为东格林威治）
`is_dst`	`boolean`	如果当前正处于夏令时范围则为真

47.72. pg_user

pg_user 提供了对数据库用户的相关信息的访问。 这个视图只是一个 pg_shadow 的公众可读的部分的视图化， 它把口令域给刷掉了。

Table 47-73. pg_user 字段

名字	类型	描述
usename	name	用户名
usesysid	oid	用户 ID
usecreatedb	bool	用户可以创建数据库
usesuper	bool	用户是一个超级用户
usecatupd	bool	用户可以更新系统表。即使超级用户也不能这么干， 除非这个字段为真。
userepl	bool	用户可以初始化流复制并且使系统处于或离开备份模式。
passwd	text	不是口令(总是为 ********)
valuntil	abstime	口令失效的时间(只用于口令认证)
useconfig	text[]	运行时配置参数的会话缺省

47.73. `pg_user_mappings`

`pg_user_mappings` 提供访问关于用户映射的信息的接口。 这个视图只是一个 `pg_user_mapping` 的公众可读的部分的视图化，如果用户无权使用它则空着选项字段。

Table 47-74. `pg_user_mappings` 字段

名字	类型	引用	描述
`umid`	`oid`	`pg_user_mapping`.oid	用户映射的OID
`srvid`	`oid`	`pg_foreign_server`.oid	包含这个映射的外部服务器的OID
`srvname`	`name`	`pg_foreign_server`.srvname	外部服务器的名字
`umuser`	`oid`	`pg_authid`.oid	被映射的本地角色的OID，如果用户映射是公共的则为0
`usename`	`name`	被映射的本地用户的名字	
`umoptions`	`text[]`	如果当前用户是外部服务器的所有者，则为用户映射指定选项， 使用"keyword=value"字符串，否则为null	

47.74. `pg_views`

`pg_views` 提供了对数据库里每个视图的有用信息的访问途径。

Table 47-75. `pg_views` 字段

名字	类型	引用	描述
`schemaname`	`name`	`pg_namespace`.nspname	包含此视图的模式名字
`viewname`	`name`	`pg_class`.relname	视图的名字
`viewowner`	`name`	`pg_authid`.rolname	视图的所有者的名字
`definition`	`text`	视图定义(一个重建的 SELECT 查询)	

Chapter 48. 前/后端协议

Table of Contents

PostgreSQL使用一种基于消息的协议用于前端和后端(服务器和客户机)之间通讯。 该协议是在 TCP/IP 和 Unix 域套接字上实现的。 端口号 5432 已经在 IANA 注册为使用这种协议的常用端口，但实际上任何非特权端口号都可以使用。

这份文档描述了版本 3.0 的协议，在 PostgreSQL 7.4 和以后的版本中实现。 对于以前的协议的描述，请参考以前版本的 PostgreSQL 文档。 一个服务器可以支持多种协议版本。初始化的启动消息告诉服务器客户端可以接受的协议版本，然后服务器则遵守该版本的协议(如果它能做到的话)。

为了可以有效地为多个客户端提供服务，服务器为每个客户端派生一个新的"后端"进程。 在目前的实现里，在检测到到来的连接请求后，马上创建一个新的子进程。 不过，这些是对协议透明的。对于协议而言，术语"后端"和"服务器"是可以互换的；类似的还有"前端"和"客户端"也是可以互换的。

48.1. 概要

协议在启动和正常操作过程中有不同的阶段。 在启动阶段里，前端打开一个到服务器的连接并且认证自身以满足服务器。 这些可能包含一条消息，也可能包含多条消息，根据使用的认证方法而不同。 如果所有这些事情都运行平稳，那么服务器就发送状态信息给前端，并最后进入正常操作。 除了初始化的启动请求之外，这部分协议是服务器驱动的。

在正常操作中，前端发送查询和其它命令到后端，然后后端返回查询结果和其它响应。 有少数几种情况(比如 `NOTIFY`)是后端发送主动提供的消息，但绝大多数情况下会话都是由前端请求驱动的。

会话的终止通常是由前端来选择的，但是也可以在某些情况下由后端强制执行。 不管那种情况，如果后端关闭连接，那么他将在退出之前回滚(完成)所有打开的(未完成的)事务。

在正常操作中，SQL 命令可以通过两个子协议中的任何一个执行。 在"简单查询"协议中，前端只是发送一个文本查询串，然后后端马上解析并执行它。 在"扩展查询"协议中，查询的处理被分割为多个步骤：解析，参数值绑定，和执行。 这样就可以提供灵活性和性能的改进，代价是额外的复杂性。

正常操作有用于类似 `COPY` 这样的额外的子协议。

48.1.1. 消息概述

所有通讯都是通过一个消息流进行的。 消息的第一个字节标识消息类型，然后后面跟着的四个字节声明消息剩下部分的长度(这个长度包括长度域自身，但是不包括消息类型字节)。 剩下的消息内容由消息类型决定。由于历史原因，客户端发送的最早的消息(启动消息)没有初始的消息类型字节。

为了避免和消息流失去同步，服务器和客户端通常都是把整个消息读取到一个缓冲区里(使用字节计数)，然后才试图处理其内容。 这样在处理内容的过程中，如果发现错误，就比较容易恢复。 在非常罕见的情况下(比如说没有足够的内存用来缓冲消息)，接收端可以使用字节计数来判断它在重新读取消息之前需要忽略多少输入。

通常，服务器和客户端都需要注意决不发送一条不完整的消息。 这些通常是通过在发送消息之前，在一个缓冲区里整理整条消息。 如果在发送或者接受一条消息的中间发生了通讯错误，那么唯一合理的反应是放弃连接，因为能够恢复消息边界同步的可能性很小。

48.1.2. 扩展查询概述

在扩展查询协议中，SQL 命令的执行是被分割成多个步骤的。 步骤与步骤之间保存的状态是由两类的对象代表的：预备语句和入口。 一个预备语句代表一个文本查询字符串的经过语法分析，语意分析，以及规划之后的结果。 一个预备语句不一定就是可以执行的，因为它可能还缺乏参数的值。 一个入口代表一个已经可以执行的或者已经部分执行过的语句，所有参数都已经填充到位了。 (对于 `SELECT` 语句，入口等效于一个打开的游标，使用不同的术语是因为游标不能处理非 `SELECT` 语句)。

完整的执行周期包括一个解析步骤，它用一个文本的查询字符串创建一个预备语句； 一个绑定步骤，它用一个预备语句和任何所需要的参数值创建一个入口； 以及一个执行步骤，它运行一个入口的查询。 如果是一个返回数据行的查询(`SELECT` , `SHOW` 等)，可以告诉执行步骤只抓取有限的一些行，这样就可能需要多个执行步骤来完成操作。

后端可以跟踪多个预备语句和入口(但是要注意，这些只在一个会话内部存在，从来不能在会话之间共享)。 现存的预备语句和入口都是用创建它们的时候赋予的名字引用的。 另外，还存在一个"未命名"的预备语句和入口。 尽管它们的行为和命名对象大部分相同，但是它们是针对只执行一次然后就抛弃的查询进行优化过的，而在命名对象上的操作是针对多次使用优化的。

48.1.3. 格式和格式代码

特定数据类型的数据可以用几种不同的格式中的任意一种来传递。 到 PostgreSQL 7.4 的时候，只支持"文本"和"二进制"两种格式，但是协议为未来的扩展提供了的手段。 任意值要求的格式是用一个格式代码声明的。客户端可以为每个传输的参数值和查询结果的每个字段声明一个格式代码。 文本的格式代码是0，二进制的格式代码是1，所有其它的格式代码都保留给将来定义。

文本方式的数值是特定数据类型的输入/输出转换函数接受或者生成的数值的字符串形式。 在传输时的表现上，字符串末尾没有空字符；如果前端要想把收到的值当作 C 字符串处理，那么必须自己加上一个。 (另外，文本格式不允许嵌入的空。)

整数的二进制表现形式采用网络字节序(高位在前)。 对于其它数据类型，请参考文档或者源代码了解其二进制表现形式。 请注意，复杂的数据类型的二进制形式可能在不同服务器版本之间变化；文本格式通常是最具有移植性的选择。

48.2. 消息流

本节描述消息流，以及每种消息类型的语意(每种消息的准确表现细节在Section 48.5里)。 因连接的状态不同，存在几种不同的子协议：启动、查询、函数调用、 `COPY` 、结束。 还有用于异步操作(包括通知响应和命令取消)的特殊规定，这些异步操作可能在启动阶段过后的任何时间产生。

48.2.1. 启动

要开始一个会话，前端打开一个与服务器的连接并且发送一个启动消息。 这个消息包括用户名以及用户希望与之连接的数据库；它还标识要使用的特定的协议版本。 (另外，启动消息可以包括用于运行时参数的额外设置。) 服务器然后就使用这些信息以及它的配置文件的内容(比如 `pg_hba.conf`)以判断这个连接是否可以接受，以及需要什么样的额外的认证(如果有)。

然后服务器就发送合适的认证请求消息，前端必须用合适的认证响应消息来响应(比如一个口令)。 对所有的认证方法，除了GSSAPI和SSPI，最多只有一次请求和响应。 有些认证方法则根本不需要前端的响应，因此就没有认证请求发生。 对于GSSAPI和SSPI，为了完成认证则可能需要多次数据包的交换。

认证周期的结束要么是服务器拒绝连接(ErrorResponse)，要么是服务器发送AuthenticationOk消息。

这个阶段来自服务器可能消息是：

ErrorResponse

连接请求被拒绝。然后服务器马上关闭连接。

AuthenticationOk

认证信息的交换成功完成。

AuthenticationKerberosV5

现在前端必须与服务器进行一次 KerberosV5 认证对话(是Kerberos规范的一部分，在这里没有描述)。 如果对话成功，服务器响应一个 AuthenticationOk消息，否则它响应一个ErrorResponse消息。

AuthenticationCleartextPassword

现在前端必须发送一个包含明文口令的 PasswordMessage包。 如果这是正确的口令，服务器用一个 AuthenticationOk 包响应，否则它响应一个 ErrorResponse 包。

AuthenticationMD5Password

现在前端必须发送一个包含用 MD5 加密的口令（包括用户名）的 PasswordMessage ， 加密时使用在 AuthenticationMD5Password 消息里指定的 4 字节随机盐粒。 如果这是正确口令，服务器用一个 AuthenticationOk 响应，否则它用一个 ErrorResponse 响应。 实际的 PasswordMessage可以通过SQL语句 `concat('md5',md5(concat(md5(concat(password, username)), random-salt)))` 计算得到。 (注意 `md5()` 函数的返回值是16进制表示的字符串。)

AuthenticationSCMCredential

这个响应只会在那些支持 SCM 信任消息的本地 Unix 域连接上出现。 前端必须发出一条 SCM 信任消息然后发送一个数据字节。 （数据字节的内容并不会被关心；它的作用只是确保服务器等待了足够长的时间来接受信任信息。） 如果信任是可以接受的，那么服务器用 AuthenticationOk 响应，否则用 ErrorResponse 响应。 （只有9.1以前的服务器才可能发出这个消息，并且这个消息最终可能会被从协议中删除。）

AuthenticationGSS

现在前端必须发起一个GSSAPI协商。 作为响应前端将随着GSSAPI数据流的最初部分发送一个PasswordMessage包。 如果需要进一步的消息，服务端将以AuthenticationGSSContinue作为响应。

AuthenticationSSPI

现在前端必须发起一个SSPI协商。 作为响应前端将随着SSPI数据流的最初部分发送一个PasswordMessage包。 如果需要进一步的消息，服务端将以AuthenticationGSSContinue作为响应。

AuthenticationGSSContinue

这个消息包含GSSAPI或SSPI协商的上一个步骤（AuthenticationGSS, AuthenticationSSPI或前一个AuthenticationGSSContinue）的响应数据。 如果这个消息中的GSSAPI或SSPI数据指示需要更多的数据以完成认证过程，前端必须用另一个PasswordMessage包发送这些数据。 如果通过这个消息GSSAPI或SSPI认证已完成，服务端下次将发送AuthenticationOk指示认证成功或ErrorResponse指示认证失败。

如果前端不支持服务器要求的认证方式，那么它应该马上关闭连接。

在收到 AuthenticationOk 包之后，前端必须等待来自后端的更多消息。 在这个阶段会启动一个后端进程，而前端只是一个感兴趣的看热闹的。 启动尝试仍然有可能失败 (ErrorResponse)，但是通常情况下，后端将发送一些 ParameterStatus 消息、 BackendKeyData 、最后是 ReadyForQuery 。

在这个阶段，后端将尝试应用任何在启动消息里给出的额外的运行时参数设置。 如果成功，这些值将成为会话的缺省值。错误将导致 ErrorResponse 并退出。

这个阶段来自后端的可能消息是：

BackendKeyData

这个消息提供了密钥(secret-key)数据，前端如果想要在稍后发出取消的请求，则必须保存这个数据。 前端不应该响应这个消息，但是应该继续侦听等待 ReadyForQuery 消息。

ParameterStatus

这个消息告诉前端有关后端参数的当前(初始化)设置，比如 client_encoding 或 DateStyle 等。 前端可以忽略这个消息，或者记录其设置用于将来使用；参阅Section 48.2.6获取更多细节。 前端不应该响应这个消息，而是应该继续侦听 ReadyForQuery 消息。

ReadyForQuery

后端启动成功，前端现在可以发出命令。

ErrorResponse

后端启动失败。在发送完这个消息之后连接被关闭。

NoticeResponse

发出了一个警告消息。前端应该显示这个消息，并且继续等待 ReadyForQuery 或 ErrorResponse 。

后端在每个查询循环后都会发出一个相同的 ReadyForQuery 消息。 前端可以合理地认为 ReadyForQuery 是一个查询循环的开始， 或者认为 ReadyForQuery 是启动阶段和每个随后查询循环的结束，具体是哪种情况取决于前端的编码需要。

48.2.2. 简单查询

一个查询循环是由前端发送一条 Query 消息给后端进行初始化的。 这条消息包含一个用文本字符串表示的 SQL 命令(或者一些命令)。 后端根据查询命令字符串的内容发送一条或者更多条响应消息给前端，并且最后是一条 ReadyForQuery 响应消息。 ReadyForQuery 通知前端它可以安全地发送新命令了。 (实际上前端不必在发送其它命令之前等待 ReadyForQuery，但是这样一来，前端必须负责区分早先发出的命令失败，而稍后发出的命令成功的情况。)

从后端来的可能的消息是：

CommandComplete

一个 SQL 命令正常结束

CopyInResponse

后端已经准备好从前端拷贝数据到一个表里面去。又见Section 48.2.5。

CopyOutResponse

后端已经准备好从一个表里拷贝数据到前端里面去。又见Section 48.2.5。

RowDescription

表示为了响应一个 `SELECT`，`FETCH` 等的查询，将要返回一个行。 这条消息的内容描述了这行的字段布局。这条消息后面将跟着每个返回给前端的行一个的 DataRow 消息。

DataRow

`SELECT`，`FETCH` 等查询返回的结果集中的一行。

EmptyQueryResponse

识别了一个空的查询字符串。

ErrorResponse

出错了。

ReadyForQuery

查询字符串的处理完成。发送一个独立的消息来标识这个是因为查询字符串可能包含多个 SQL 命令。 (CommandComplete 只是标记一条 SQL 命令处理完毕，而不是整个字符串。) ReadyForQuery 总会被发送，不管是处理成功结束还是产生错误。

NoticeResponse

发送了一个与查询有关的警告消息。注意警告消息是附加在其它响应上的，也就是说,后端将继续处理该命令。

`SELECT`(或其它返回结果集的查询，比如 `EXPLAIN` 或 `SHOW`)查询的响应通常包含 RowDescription， 零个或者多个 DataRow 消息，以及最后的 CommandComplete 。 从或者到前端的 `COPY` 使用Section 48.2.5里提到的特殊的协议。所有其它查询类型通常只生成一个 CommandComplete 消息。

因为查询字符串可能包含若干个查询(用分号分隔)，所以在后端完成查询字符串的处理之前可能有好几个这样的响应序列。 如果整个字符串已经处理完，后端已经准备好接受新查询字符串的时候则发出 ReadyForQuery 消息。

如果收到一个完全空(除了空白之外没有内容)的查询字符串，那么响应是一条 EmptyQueryResponse 后面跟着 ReadyForQuery 。

在出现错误的时候，发出一个 ErrorResponse 消息，后面跟着 ReadyForQuery 。 查询字符串的所有后继的处理都被 ErrorResponse 中止(即使里面还有查询也这么干)。 请注意这些事情可能在处理一个查询产生的消息序列的中途发生。

在简单查询模式，检索出来的数值的格式总是文本的，除非给出的命令是从一个声明了 `BINARY` 选项的游标上的 `FETCH` 。 在这种情况下，检索出来的数值是二进制格式的。在 RowDescription 消息里给出的格式代码告诉用了哪种格式。

前端在等待其它类型的消息时必须准备接收 ErrorResponse 和 NoticeResponse 消息。 参阅Section 48.2.6后端因为外部的事件可能生成的消息。

建议的方法是把前端代码写成状态机的风格，它可以在任何时刻接受任何有意义的消息，而不是写成假设消息的准确序列的代码。

48.2.3. 扩展查询

扩展的查询协议把上面描述的简单协议分裂成若干个步骤。准备的步骤可以多次复用以提高效率。 另外，还可以获得额外的特性，比如把数据值作为独立的参数提供的可能性，而不是把它们直接插入一个查询字符串。

在扩展的协议里，前端首先发送一个 Parse 消息，它包含一个文本查询字符串，另外还有一些有关参数占位符的数据类型的信息，以及一个最终预备语句对象的名字(一个空字符串选择未命名的预备语句)。 响应要么是一个 ParseComplete 要么是 ErrorResponse 。参数数据类型可以用 OID 来声明；如果没有给出，那么分析器将试图用它对付无类型的字符串常量的方法来推导其数据类型。

> **Note:** 一个参数数据类型可以通过设置为零，或者让参数类型 OID 的数目比查询字符串里的参数符号(`$``_n_` )的数目少的方法不予声明。 另外一个特例是参数的类型可以声明为 `void` (也就是伪类型 `void` 的 OID)。 这是为了允许用于某些函数参数的参数符号实际上是 OUT 参数。 通常情况下，没有什么环境会用到 `void` 参数，但是如果在函数的参数列表里出现了这么一个参数符号，那么它实际上会被忽略。 比如，一个像这样的函数调用： `foo($1,$2,$3,$4)` ， 如果 `$3` 和 `$4` 声明为类型是 `void` ， 那么这个函数调用可以匹配一个带有两个 IN 和两个 OUT 参数的函数。
>
> **Note:** 在一个 Parse 消息里包含的查询字符串不能包含超过一个 SQL 语句；否则就会报告一个语法错误。 这个限制在简单查询协议中并不存在，但是它存在于扩展的协议中，因为允许预备语句或者入口包含多个命令将导致协议过度地复杂。

如果成功创建了一个命名的预备语句对象，那么它将持续到当前会话结束，除非明确删除。一个未命名的预备语句只持续到下一个声明未命名的语句的 Parse 语句发出为止(请注意一个简单的查询消息也删除未命名语句)。 命名的预备语句必须明确地关闭，然后才能用一个 Parse 消息重新定义，但是未命名的语句并不要求这个动作。 命名的预备语句也可以在 SQL 命令级创建和访问，方法是使用 `PREPARE` 和 `EXECUTE` 。

只要预备语句还存在，那么就可以使用 Bind 消息很容易地使之进入执行状态。 Bind 消息给出源预备语句的名字(空字符串表示未命名的预备语句)，目标入口的名字(空字符串表示未命名的入口)，以及用于那些在预备语句中出现的所有参数占位符的数值。 提供的参数集必须匹

配那些预备语句需要的东西。(如果你在 Parse 消息里声明任何 `void` 参数，那么在 Bind 消息里给它们传递 NULL 值。) Bind 还声明用于查询返回的任何数据的格式；格式可以一次声明，也可以每个字段进行声明。响应要么是 BindComplete 要么是 ErrorResponse 。

> **Note:** 输出的格式是文本还是二进制是由 Bind 里给出的格式代码决定的，不管用的是什么 SQL 命令。 在使用扩展的查询协议的时候，游标声明里的 `BINARY` 属性是不起作用的。

典型的查询计划在处理Bind消息后生成。 预备语句如果不包含参数，或者被反复执行的情况下，服务端可能会保存创建好的查询计划并在后续的针对同样的Bind消息中重用它。 但是，只有确保这样作成的通用的查询计划不至于比依赖于特定参数值的查询计划效率差太多的情况下，服务端才会这么做。 就协议而言这些是透明的。

如果成功创建了一个命名的入口对象，那么它将持续到当前会话结束，除非明确删除。 一个未命名的入口只持续到事务结束或下一个声明未命名的入口的 Bind消息发出为止(请注意一个简单的查询消息也删除未命名入口)。 命名的入口必须明确地关闭，然后才能用另一个 Bind 消息重新定义，但是未命名的语句并不要求这个动作。 命名的入口也可以在 SQL 命令级创建和访问，方法是使用`DECLARE CURSOR` 和 `FETCH` 。

只要存在一个入口，那么就可以用一个 Execute 消息执行它。 Execute 消息声明入口的名字(空字符串表示未命名入口)和一个最大的结果行计数(零表示"抓取所有行")。 结果行计数只对包含返回结果集的入口有意义；在其它情况下，该命令总是执行到结束，而行计数会被忽略。 Execute 可能的响应和那些通过简单查询协议发出的查询一样，只不过 Execute 不会导致后端发出 ReadyForQuery 或者 RowDescription 。

如果 Execute 在一个入口的执行完成之前终止(因为达到了一个非零的结果行计数)，它将发送一个 PortalSuspended 消息；这个消息的出现告诉前端应该在同一个入口上发出另外一个 Execute 以完成操作。 在入口的执行完成之前，不会发出表示源 SQL 命令结束的 CommandComplete 消息。 因此 Execute 阶段的结束总是由出现下列之一的消息标志的：CommandComplete 、EmptyQueryResponse(如果入口是从一个空字符串创建出来的)、ErrorResponse 、PortalSuspended 。

每个扩展查询消息序列完成后，前端都应该发出一条 Sync 消息。 这个无参数的消息导致后端关闭当前事务——如果当前事务不是在一个 `BEGIN` / `COMMIT` 事务块中的话("关闭"的意思就是在没有错误的情况下提交，或者是有错误的情况下回滚)。 然后响应一条 ReadyForQuery 消息。Sync 的目的是提供一个错误恢复的重新同步的点。 如果在处理任何扩展查询消息的时候侦测到任何错误，那么后端发出 ErrorResponse ，然后读取并抛弃消息，直到一个 sync 的到来，然后发出 ReadyForQuery 并且返回到正常的消息处理中。 (但是要注意如果正在处理 Sync 的时候发生了错误，那么不会忽略任何东西 —这样就保证了为每个 Sync 发出一个并且只有一个的 ReadyForQuery 。

> **Note:** Sync 并不导致一个用 `BEGIN` 打开的事务块关闭。可以侦测到这种情况，因为 ReadyForQuery 消息包含事务状态信息。

除了这些基本的，必须的操作之外，在扩展查询协议里还有几种可选的操作可以使用。

Describe 消息(入口变体)指定一个现有的入口的名字(如果是一个未命名的入口则是一个空字符串)。 响应是一个 RowDescription 消息，它描述了执行入口将要返回的行；或者是一个 NoData 消息(如果入口并不包含会返回行的查询)；或者是一个 ErrorResponse(如果没有这个入口)。

Describe 消息(语句变体)指定一个现有的预备语句的名字(如果是一个未命名的预备语句则是一个空字符串)。 响应是一个描述该语句需要的参数的 ParameterDescription 消息，后面跟着一个描述语句最终执行后返回的行的 RowDescription 消息(或者是 NoData 消息，如果该语句不返回行)。 如果没有这样的预备语句，则返回 ErrorResponse 。 请注意因为还没有发出 Bind ， 所以后端还不知道用于返回字段的格式；在这种情况下，RowDescription 消息里面的格式代码字段将是零。

> **Tip:** 在大多数情况下，前端在发出 Execute 之前应该发出某种 Describe 的变体，以保证它知道如何解析它将得到的结果。

Close 消息关闭一个现有的预备语句或者入口，并且释放资源。 对一个不存在的语句或者入口名字发出 Close 不是一个错误。 响应通常是 CloseComplete ，但如果在释放资源的时候发生了一些困难也可以是 ErrorResponse 。 请注意关闭一个预备语句隐含地关闭任何从该语句构造出来的打开的入口。

Flush 消息并不导致任何特定的输出的生成，但是强制后端发送任何还在它的输出缓冲区中停留的数据。 Flush 必须在除 Sync 外的任何扩展查询命令后面发出(如果前端希望在发出更多的命令之前检查该命令的结果的话)。 如果不 Flush ，后端返回的消息将组合成最小可能的数据包个数，以减少网络负荷。

> **Note:** 简单查询消息大概等于一系列使用未命名的预备语句和无参数的入口对象的 Parse 、Bind 、入口 Describe 、Execute 、Close 、Sync 。 一个区别是它会在查询字符串中接受多个 SQL 语句，并在会话中为每个语句自动执行绑定/描述/执行序列。 另外一个区别是它不会返回 ParseComplete 、Bindcomplete 、CloseComplete 、NoData 消息。

48.2.4. 函数调用

函数调用子协议允许客户端请求一个对存在于数据库 `pg_proc` 系统表中的任意函数的直接调用。 客户端必须在该函数上有执行的权限。

> **Note:** 函数调用子协议是一个遗留的特性，在新代码里最好避免用它。 类似的结果可以通过设置一个 `SELECT function($1, ...)` 预备语句获得。 这样函数调用循环就可以用 Bind/Execute 代替。

一个函数调用循环是由前端向后端发送一条 FunctionCall 消息初始化的。 然后后端根据函数调用的结果发送一条或者更多响应消息，并且在最后是一条 ReadyForQuery 响应消息。 ReadyForQuery 通知前端它可以安全地发送一条新的查询或者函数调用了。

从后端来的可能的消息是：

ErrorResponse

发生了一个错误。

FunctionCallResponse

函数调用完成并且在消息中返回一个结果。(请注意，函数调用协议只能处理单个标量结果，不能处理行类型或者结果集。)

ReadyForQuery

函数调用处理完成。ReadyForQuery 将总是被发送，不管是成功完成处理还是发生了一个错误。

NoticeResponse

发出了一条有关该函数调用的警告消息。通知是附加在其它响应上的，也就是说，后端将继续处理命令。

48.2.5. COPY操作

`COPY` 命令允许在服务器和客户端之间高速的大批量数据传输。 拷贝入和拷贝出操作每个都把连接切换到一个独立的子协议中，并且持续到操作结束。

拷贝入模式(数据传输到服务器)是在后端执行一个 `COPY FROM STDIN` 语句的时候初始化的。 后端发送一个 CopyInResponse 消息给前端。 前端应该发送零条或者更多 CopyData 消息，形成一个输出数据的流。 (消息的边界和行的边界没有任何相关性要求，尽管通常那就是合理的边界选择。) 前端可以通过发送一个 CopyDone 消息来终止拷贝入操作(允许成功终止)，也可以发出一个 CopyFail 消息(它将导致 `COPY` 语句带着错误失败)。 然后后端就恢复回它在 `COPY` 开始之前的命令处理模式，可能是简单查询协议，也可能是扩展查询协议。 然后它会发送 CommandComplete(如果成功)或者 ErrorResponse(如果失败)。

如果在拷贝入模式下后端检测到了错误(包括接受接收到 CopyFiail 消息，或者是任何除了 CopyData 或者 CopyDone 之外的前端消息)，那么后端将发出一个 ErrorResponse 消息。 如果 `COPY` 命令是通过一个扩展的查询消息发出的，那么后端从现在开始将抛弃前端消息，直到一个 Sync 消息到达，然后它将发出 ReadyForQuery 并且返回到正常的处理中。 如果 `COPY` 命令是在一个简单查询消息里发出的，那么该消息剩余部分被丢弃然后发出 ReadyForQuery 消息。 不管是哪种情况，任何前端发出的 CopyData, CopyDone, CopyFail 消息都将被简单地抛弃。

后端将会忽略拷贝入模式时接收到的 Flush 和 Sync 消息。 收到任何其它非拷贝消息，会引发错误，如前所述这样会导致退出拷贝入状态。 (Flush 和 Sync 的例外是为了方便客户端库在执行 Execute 消息后始终发送 Flush 或 Sync 而不检查所执行的命令是否是一个 COPY

FROM STDIN 命令。)

拷贝出模式(数据从服务器发出)是在后端执行一个 `COPY TO STDOUT` 语句的时候初始化的。 后端发出一个 CopyOutResponse 消息给前端，后面跟着零或者多个 CopyData 消息(总是每行一个)，然后跟着 CopyDone 。 然后后端回退到它在 `COPY` 开始之前的命令处理模式，然后发送 CommandComplete 。 前端不能退出传输(除非是关闭连接或者发出一个 Cancel 请求)，但是它可以抛弃不需要的 CopyData 和 CopyDone 消息。

在拷贝出模式中，如果后端检测到错误，那么它将发出一个 ErrorResponse 消息并且回到正常的处理。 前端应该把收到 ErrorResponse当作终止拷贝出模式的标志。

NoticeResponse和ParameterStatus消息有可能夹在CopyData消息间出现。 前端必须处理这种情况，并且也应该准备好处理其它异步消息(参阅Section 48.2.6)。 除此以外，任何CopyData和CopyDone以外的消息应该被视作拷贝出模式的中止。

还有一个拷贝相关的模式，称作双向拷贝，它允许从和到服务端的高速的批量数据传输。 双向拷贝模式由处于walsender模式的后端执行一条 `START_REPLICATION` 语句初始化。 后端发送一个CopyBothResponse给前端。然后后端和前端可能都发送CopyData消息，直到有一方发出一个CopyDone消息。 客户端发送CopyDone消息后，连接从双向拷贝模式切换到拷贝出模式，并且客户端不应该再发出任何CopyData消息。 相似的，如果服务端发出CopyDone消息，连接进入拷贝入模式，并且服务端不应该再发出任何CopyData消息。 两边都发送了CopyDone消息后，拷贝模式终止，后端返回到命令处理模式。 在双向拷贝模式中，如果后端检出错误，将发出一个ErrorResponse消息，丢弃任何来自前端的消息直到收到Sync消息，然后发送ReadyForQuery消息并返回到普通的处理模式。 前端应把收到ErrorResponse消息作为在两个方向上都终止拷贝的标志，这个时候不应该发出CopyDone消息。 请参阅Section 48.3获得更多有关双向拷贝模式下子协议传输的信息。

CopyInResponse，CopyOutResponse和 CopyBothResponse 消息包含了告诉前端每行的字段数以及每个字段使用的格式代码的信息。 (就目前的实现而言，某个 `COPY` 操作的所有字段都使用同样的格式，但是消息设计并不做这个假设。)

48.2.6. 异步操作

有几种情况下后端会发送一些并非由特定的前端的命令流提示的消息。 在任何时候前端都必须准备处理这些消息，即使是并未涉及到查询处理的时候。 至少，应该在开始读取查询响应之前检查这些情况。

NoticeResponse 消息有可能是因为外部的活动而生成的；比如，如果数据库管理员进行一次"快速"数据库关闭，那么后端将在关闭连接之前发送一个 NoticeResponse 来通知这件事。因此，前端应该总是准备接受和显示 NoticeResponse 消息，即使连接表面上是空闲的时候也如此。

如果后端认为前端应该知道的任何参数的实际值发生了变化，那么都会产生 ParameterStatus 消息。 这些最经常发生的地方是前端执行的一个 `SET` 命令的响应，并且这个时候实际上是同步(但是也有可能是数据库管理员改变了配置文件然后SIGHUP了服务器导致的参数状态的变化)。 同样，如果一个 `SET` 命令回滚，那么也会生成合适的 ParameterStatus 消息以报告当前有效的数值。

目前，系统内有一套会生成 ParameterStatus 消息的硬编码的参数：他们是 `server_version` 、 `server_encoding` 、 `client_encoding` 、 `application_name` 、 `is_superuser` 、 `session_authorization` 、 `DateStyle` 、 `IntervalStyle` 、 `TimeZone` 、 `integer_datetimes` 以及 `standard_conforming_strings` 。 （8.0以前的版本不会报告 `server_encoding` 、 `TimeZone` 和 `integer_datetimes` 。 8.1以前的版本不会报告 `standard_conforming_strings` 。 8.4以前的版本不会报告 `IntervalStyle` 。 9.0以前的版本不会报告 `application_name` 。） 请注意 `server_version` 、 `server_encoding` 和 `integer_datetimes` 是伪参数，启动后不能修改。 这个参数集合可能在将会改变，或者甚至是变成可以配置的。因此，前端应该简单地忽略那些它不懂或者不关心的 ParameterStatus 。

如果前端发出一个 `LISTEN` 命令，那么为同一个通道名执行了 `NOTIFY` 命令后，将收到后端发来的一个 NotificationResponse 消息(不要和 NoticeResponse 混淆)。

> **Note:** 目前，NotificationResponse 只能在一个事务外面发送，因此它将不会在一个命令响应序列中间出现，但是它可能在 ReadyForQuery 之前出现。 不过，在前端逻辑中做上述假设是不明智的。好的做法是在协议的任何点上都可以接受 NotificationResponse 。

48.2.7. 取消正在处理的请求

在一条查询正在处理的时候，前端可能请求取消这个查询。 这样的取消请求不是直接通过打开的当前连接发送给后端的，这么做是因为实现的有效性：不希望后端在处理查询的过程中不停地检查前端来的输入。 取消请求应该相对而言比较少见，所以把取消做得稍微笨拙一些，以便不影响正常状况的性能。

要发送一条取消请求，前端打开一个与服务器的新连接并且发送一条 CancelRequest 消息，而不是通常在新连接中经常发送的 StartupPacket 消息。 服务器将处理这个请求然后关闭连接。出于安全原因，对取消请求消息不做直接的响应。

除非 CancelRequest 消息包含与连接启动过程中传递给前端的相同的键数据(PID 和安全键字)，否则它将被忽略。 如果该请求匹配当前运行着的后端的 PID 和安全键字，则退出当前查询的处理(目前的实现里采用的方法是向正在处理该查询的后端进程发送一个特殊的信号)。

取消信号可能有也可能没有任何作用。 例如，如果它在后端已经完成了查询的处理后到达，那么它就没有作用。 如果取消起作用了，其结果是当前命令带着一个错误消息提前退出。

这么做是对安全和有效性通盘考虑的结果，前端没有直接的方法获知一个取消请求是否成功。 它必须继续等待后端对查询响应。执行取消仅仅是增加了当前查询快些结束的可能性，以及增加了当前查询会带着一条错误消息失败而不是成功执行的可能性。

因为取消请求是通过新的连接发送给服务器而不是通过通常的前/后端通讯连接，所以取消请求可能是任意进程执行的，而不仅仅是要取消查询的前端。 这样可能对创建多进程应用有某种灵活性的好处。但是同时这样也带来了安全风险，因为这样任何一个非认证用户都可能试图取消查询。 这个安全风险通过要求在取消请求中提供一个动态生成的安全键字排除。

48.2.8. 终止

通常，优雅的终止过程是前端发送一条 Terminate消息并且立刻关闭连接。一旦收到这个消息，后端马上关闭连接并且退出。

在少数情况下(比如通过一个管理员命令关闭数据库)，后端可能在没有任何前端请求的情况下断开连接。 在这种情况下，后端将在它断开连接之前尝试发送一个错误或者通知消息，给出断开的原因。

其它终止的情况发生在各种失效的场合，比如某一方的内核转储，失去通讯链路，丢失了消息边界同步等。 不管是前端还是后端看到了一个意外的连接关闭，那么它应该清理现场并且终止。 如果前端不想终止自己，那么它可以通过再次连接服务器的方式重启一个新的后端。 如果收到了一个无法识别的消息，那么也建议关闭连接，因为出现这种情况可能意味着是丢失了消息边界的同步。

不管是正常还是异常的终止，任何打开的事务都会回滚，而不是提交。 不过，应该注意的是如果一个前端在一个非 `SELECT` 查询正在处理的时候断开，那么后端很可能在注意到断开之前先完成查询的处理。 如果查询处于任何事务块之外((`BEGIN` ... `COMMIT` 序列)，那么其结果很可能在得知断开之前被提交。

48.2.9. SSL会话加密

如果编译PostgreSQL的时候打开了SSL支持，那么前后端通讯就可以用SSL加密。 这样就提供了一种在攻击者可能捕获会话通讯数据包的环境下保证通讯安全的方法。 有关使用SSL加密PostgreSQL会话的更多信息，请参阅Section 17.9。

要开始一次SSL加密连接，前端先是发送一个 SSLRequest 消息，而不是 StartupMessage 。 然后服务器以一个包含 `S` 或 `N` 的字节响应，分别表示它愿意还是不愿意进行SSL。 如果前端对响应不满意，那么它可以关闭连接。 要在 `S` 之后继续，那么先进行与服务器的SSL启动握手(没有在这里描述，这是SSL规范的一部分)。 如果这些成功了，那么继续发送普通的 StartupMessage 。 这种情况下，StartupMessage 和所有随后的数据都将由SSL加密。 要在 `N` 之后继续，则发送普通的 StartupMessage 然后不带加密进行处理。

前端应该也准备处理一个来自服务器的给 SSLRequest 的 ErrorMessage 响应。 这种情况只有在那些SSL支持被加入到PostgreSQL以前的服务上才会出现。 (这样的服务器太古老了，很可能根本就不存在。) 在这种情况下，连接必需关闭，但是前端可以选择打开一个新的连接然后不带SSL进行连接。

一个初始化的 SSLRequest 也可以用于为了发送一条 CancelRequest 消息而打开的连接中。

如果协议本身并未提供某种方法强制SSL加密，那么管理员可以把服务器配置为拒绝未加密的会话，这是认证检查的一个副产品。

48.3. 流复制协议

为了初始化流复制，前端需要发送带 `replication` 参数的startup消息。 这告诉后端进入walsender模式，在这个模式下可以发送数量不多的复制命令集而不是通常的SQL命令。 并且在walsender模式下，只能使用简单查询协议。 walsender模式下可以接受的命令如下:

IDENTIFY_SYSTEM

请求服务端标识自己。服务端会应答一个只有一行的结果集，包含3个字段。

systemid

标识数据库集群的唯一的系统标识符。 这个可以用于检查初始化备机用的基础备份来自同一个数据库集群。

timeline

当前的时间线ID(TimelineID)。 同样可用于检查备机和主机的一致性。

xlogpos

当前的xlog的写入位置。 可用于获得在事务日志中流从哪而开始的已知的位置信息。

TIMELINE*HISTORY `_tli*`

请求服务端为时间线 `_tli_` 发送时间线历史文件。 服务端会应答一个只有一行的结果集，包含3个字段。

filename

时间线历史文件的文件名，比如00000002.history。

content

时间线历史文件的内容。

START*REPLICATION `_XXX/XXX* `TIMELINE` *tli*`

指示服务端从时间线 `_tli_` 上的WAL位置 `_XXX/XXX_` 开始WAL流。 服务端可能会回应一个错误，比如，如果在请求的WAL段已经被回收的情况下。 如果成功，服务端响应一个CopyBothResponse消息，然后开启到前端的WAL流。

如果客户端请求的时间线不是最新的，但是是服务端历史的一部分，服务端将会传送从请求的开始点开始直到服务端切换到另一个时间线为止这个时间线上的所有WAL。 如果客户端请求的流正好在旧时间线的终点上，服务端会立即响应一个CommandComplete而不进入COPY模式。

传送完这个非最新的时间线上的所有WAL后，服务端退出COPY模式结束流。 当客户端也以退出COPY模式作为应答，服务端发送一个只有一行的结果集，包含2的字段，指示在这个服务器上的下一个时间线。 第一列是下一个时间线的ID，第二列是发生切换的XLOG位置。 通常切换点是已经传送的WAL流的终点，但是在极端的情况下，服务端发送的WAL可能来自自己升级前还没有回放的旧的时间线。 最终服务端发送CommandComplete消息并准备接收新的命令。

WAL数据作为一系列的CopyData消息发送。 （这允许混合其它信息，具体而言服务端在开始流之后发生了失败可以发送ErrorResponse消息。） 每个从服务端到客户端的CopyData消息的装载数据中包含下面的格式的消息：

XLogData (B)

Byte1('w')

标识消息是一个WAL数据。

Int64

消息内的WAL数据开始点。

Int64

当前服务端上的WAL終点。

Int64

传送时服务端上的系统时间，是从2000-01-01午夜开始的微秒数。

Byte `_n_`

WAL数据流的片段。

单个的WAL记录一定不会被分割成2个XLogData消息。 当一个WAL记录跨越WAL页的边界，并且因此已经被连续的记录分割了，可以在页边界上分割。 换句话说，第一个主要的WAL记录和它后续的记录可能以不同XLogData消息传送。

主keepalive消息 (B)

Byte1('k')

标识这是一个发送者的keepalive消息。

Int64

服务端上当前的WAL终点。

Int64

传送时服务端上的系统时间，是从2000-01-01午夜开始的微秒数。

Byte1

1意味着客户端应该尽可能快的应答这个消息，以防止超时切断连接。0的意思相反。

接受进程可以在任何时间使用以下消息格式(同样作为CopyData消息的装载数据)应答发送者：

备机状态更新(F)

Byte1('r')

标识这是一个接受者的状态更新消息。

Int64

备机上接受并写入磁盘的上次的WAL字节+1的位置。

Int64

备机上刷新到磁盘的上次的WAL字节+1的位置。

Int64

备机上已经应用的上次的WAL字节+1的位置。

Int64

传送时客户端上的系统时间，是从2000-01-01午夜开始的微秒数。

Byte1

如果是1，客户端请求服务端立即应答这个消息。 这用于ping服务端以测试连接是否还健康。

热备机的反馈消息(F)

Byte1('h')

标识这是一个热备机的反馈消息。

Int64

传送时客户端上的系统时间，是从2000-01-01午夜开始的微秒数。

Int32

备机上现在的xmin。如果备机正在传送热备机反馈将不再发到这个连接上的通知，这个值可能为0。之后非零的消息可能再初始化反馈机制。

Int32

备机的当前时间戳。

BASE*BACKUP [* `LABEL` `'label'_ ] [ `PROGRESS` ] [ `FAST` ] [ `WAL` ] [ `NOWAIT`]

指示服务器开始一个基础备份的流。系统将在备份开始前自动进入备份模式，并且在备份完成后回到原来的状态。 可以接受以下选项:

`LABEL` `_'label'_`

设定备份的标签。 如果没有指定，则使用 `base backup` 。 标签的引号使用规则与standard_conforming_strings开关打开时的标准SQL字符串相同。

`PROGRESS`

请求产生进度报告时要用的信息。 这将在每个表空间的头部发回一个近似的大小，用于计算到流结束还有多少距离。 这个大小是在传输前通过一次性统计所有文件的大小获取的，可能会产生性能上的冲击 - 实际上在传送第一个流前可能会花比较长的时间。 既然数据库文件可能在备份过程中改变，这个大小只是近似的，并且在估算和实际发送文件之间可能增长和收缩。

`FAST`

请求一个快速的检查点(checkpoint)。

`WAL`

包含这个备份所必要的WAL段。将包括在备份开始和结束期间 `pg_xlog` 目录下的所有base目录tar文件。

`NOWAIT`

缺省时备份会等待最后一个需要的xlog段被归档，或者在没有启用归档的情况下发出一个警告。 指定 `NOWAIT` 可以把等待和警告无效，让客户端负责确认需要的日志是否有效。

备份开始时，服务端首先发送2个通常的结果集，然后是1个以上的CopyResponse结果。

最初的通常的结果集里，包含由2列构成的单一行的备份位置。 第一个列是XLogRecPtr形式的开始位置，第二列是对应的时间线ID。

第二个通常的结果集里各个表空间一行数据。 每行包含的字段如下:

spcoid

表空间的oid。 base目录的情况下则为 `NULL` 。

spclocation

表空间目录的完全路径。base目录的情况下则为 `NULL` 。

size

要求进度状况报告的情况下，表空间的估算容量。 没有要求的情况下为 `NULL` 。

第二个通常结果集之后被送过来的是一个以上的CopyResponse结果。 一个是PGDATA用的，其余的则是每个 `pg_default` 、 `pg_global` 以外的追加表空间都有一个。 CopyResponse结果内的数据是表空间内容的tar形式（遵照POSIX 1003.1-2008规定的"ustar交换形式"）的转储(dump)。 但是，标准规定的最后的2个零数据块被省掉了。 tar数据结束后，和开始位置的形式相同，是包含备份终了位置的最终的结果集。

为data目录和每个表空间做的tar归档包含目录下的所有文件，不管它们是PostgreSQL 的文件还是相同目录下的其它在文件。 但是以下文件被排除在外:

- `postmaster.pid`
- `postmaster.opts`
- `pg_xlog` ，包含子目录。如果备份是包含WAL执行的，将包含一个合成版的 `pg_xlog` 。它只包含对于备份工作必须的文件而没有其余的内容。

如果服务器的文件系统支持的话，所有者，组和文件模式会被设置。

所有的表空间被传送完后，发送最终的通常结果集。 这个结果集里包含单一行单一列的XLogRecPtr格式的备份结束位置。

48.4. 消息数据类型

本节描述消息里用到的基本数据类型。

Int `_n_` (`_i_`)

一个网络字节顺序的 `_n_` 位整数。 如果声明了 `_i_`，它就是将出现的确切值，否则这个数值就是一个变量。 比如 Int16 ， Int32(42)。

Int `_n_` [`_k_`]

一个 `_k_` 个 `_n_` 位整数元素的数组，每个都是以网络字节顺序存储的。 数组长度 `_k_` 总是由消息前面的字段来判断的。比如 Int16[M]

String(`_s_`)

一个(C 风格的)空结尾的字符串。对字符串没有特别的长度限制。 如果声明了 `_s_`，那么它是将出现的确切的数值，否则这个数值就是一个变量。 比如 String, String("user")。

> **Note:** 后端返回的字符串的可能长度没有预定义的限制。 所以前端必须使用良好的编码策略，使用某种可扩展的缓冲区以便能接受任何能放进内存里的东西。 如果那样做不可行，则读取全长的字符串然后抛弃不能放进你的定长缓冲区的尾部字符。

Byte `_n_` (`_c_`)

精确的 `_n_` 字节。 如果字段宽度 `_n_` 不是一个常量，那么总是可以从消息中更早的字段中判断它。 如果声明了 `_c_` 那么它是确切数值。例如 Byte2, Byte1('\n')

48.5. 消息格式

本节描述各种消息的详细格式。 每种消息都标记为它是由一个前端(F)，一个后端(B)或者两者(F &B)发送的。 请注意，尽管每条消息在开头都包含一个字节计数，消息格式也定义为可以不用参考字节计数就可以找到消息的结尾。 这样就增加了有效性检查。CopyData 消息是一个例外，因为它形成一个数据流的一部分；任意独立的 CopyData 消息可能是无法自解释的。

AuthenticationOk (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节记的消息内容长度，包括这个长度本身。

Int32(0)

声明该认证是成功的。

AuthenticationKerberosV5 (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节记的消息内容长度，包括长度自身。

Int32(2)

声明需要 Kerberos V5 认证。

AuthenticationCleartextPassword (B)

Byte1('R')

标识该消息是一条认证请求。

Int32(8)

以字节记的消息内容长度，包括长度自身。

Int32(3)

声明需要一个明文的口令。

AuthenticationMD5Password (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(12)

以字节记的消息内容的长度，包括长度本身。

Int32(5)

声明需要一个 MD5 加密的口令。

Byte4

加密口令的时候使用的盐粒。

AuthenticationSCMCredential (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(8)

以字节计的消息内容长度，包括长度本身。

Int32(6)

声明需要一个 SCM 信任消息。

AuthenticationGSS (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(8)

以字节记的消息内容的长度，包括长度本身。

Int32(7)

声明需要GSSAPI认证。

AuthenticationSSPI (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32(8)

以字节记的消息内容的长度，包括长度本身。

Int32(9)

声明需要SSPI认证。

AuthenticationGSSContinue (B)

Byte1('R')

标识这条消息是一个认证请求。

Int32

以字节记的消息内容的长度，包括长度本身。

Int32(8)

标识本消息包含GSSAPI或SSPI数据。

Byte`_n_`

GSSAPI或者SSPI认证数据。

BackendKeyData (B)

Byte1('K')

标识该消息是一个取消键字数据。 如果前端希望能够在稍后发出 CancelRequest 消息，那么它必须保存这个值。

Int32(12)

以字节记的消息内容的长度，包括长度本身。

Int32

后端的进程号(PID)

Int32

此后端的密钥

Bind (F)

Byte1('B')

标识该消息是一个绑定命令

Int32

以字节记的消息内容的长度，包括长度本身。

String

目标入口的名字(空字符串则选取未命名的入口)

String

源预备语句的名字(空字符串则选取未命名的预备语句)

Int16

后面跟着的参数格式代码的数目(在下面的 `_C_` 中说明)。 这个数值可以是0，表示没有参数，或者是参数都使用缺省格式(文本)；或者是1，这种情况下声明的格式代码应用于所有参数；或者它可以等于实际数目的参数。

Int16[`_C_`]

参数格式代码。目前每个都必须是0(文本)或者1(二进制)。

Int16

后面跟着的参数值的数目(可能为零)。这些必须和查询需要的参数个数匹配。

然后，每个参数都会出现下面的字段对：

Int32

参数值的长度，以字节记(这个长度并不包含长度本身)。 可以为0。一个特殊的情况是，-1 表示一个 NULL 参数值。 在 NULL 的情况下，后面不会跟着数值字节。

Byte `_n_`

参数值，格式是用相关的格式代码表示的。 `_n_` 是上面的长度。

在最后一个参数之后，出现下面的字段：

Int16

后面跟着的结果字段格式代码数目(下面的 `_R_` 描述)。 这个数目可以是0表示没有结果字段，或者结果字段都使用缺省格式(文本)；或者是1，这种情况下声明格式代码应用于所有结果字段(如果有的话)；或者它可以等于查询的结果字段的实际数目。

Int16[`_R_`]

结果字段格式代码。目前每个必须是0(文本)或者1(二进制)。

BindComplete (B)

Byte1('2')

标识消息为一个绑定结束标识符

Int32(4)

以字节记的消息长度，包括长度本身。

CancelRequest (F)

Int32(16)

以字节记的消息长度，包括长度本身。

Int32(80877102)

取消请求代码。选这个值是为了在高 16 位包含`1234`，低 16 位包含`5678`。（为避免混乱，这个代码必须与协议版本号不同。）

Int32

目标后端的进程号(PID)

Int32

目标后端的密钥

Close (F)

Byte1('C')

标识这条消息是一个 Close 命令。

Int32

以字节计的消息内容长度，包括长度本身。

Byte1

'`S`'关闭一个准备的语句；或者'`P`'关闭一个入口。

String

一个要关闭的预备语句或者入口的名字(一个空字符串选择未命名的预备语句或者入口)。

CloseComplete (B)

Byte1('3')

标识消息是一个 Close 完毕指示器。

Int32(4)

以字节记的消息内容的长度，包括长度本身。

CommandComplete (B)

Byte1('C')

标识此消息是一个命令结束响应。

Int32

以字节记的消息内容的长度，包括长度本身。

String

命令标记。它通常是一个单字，标识那个命令完成。

对于 `INSERT` 命令，标记是 `INSERT` `_oid_` `_rows_`，这里的 `_rows_` 是插入的行数。`_oid_` 在 `_rows_` 为1并且目标表有OID的时候是插入行的对象ID； 否则 `_oid_` 就是0。

对于 `DELETE` 命令，标记是 `DELETE` `_rows_`，这里的 `_rows_` 是删除的行数。

对于 `UPDATE` 命令，标记是 `UPDATE` `_rows_` ，这里的 `_rows_` 是更新的行数。

对于 `SELECT` 或 `CREATE TABLE AS` 命令， 标记是 `SELECT` `_rows_`， 这里的 `_rows_` 是检出的行数。

对于 `MOVE` 命令，标记是 `MOVE` `_rows_`，这里的 `_rows_` 是游标位置改变的行数。

对于 `FETCH` 命令，标记是 `FETCH` `_rows_`，这里的 `_rows_` 是从游标中检索出来的行数。

对于 `COPY` 命令，标记是 `COPY` `_rows_`，这里的 `_rows_` 是拷贝的行数。 （注意：这个行数只出现在PostgreSQL 8.2及以后的版本中）。

CopyData (F & B)

Byte1('d')

标识这条消息是一个 COPY 数据。

Int32

以字节记的消息内容的长度，包括长度本身。

Byte `_n_`

`COPY` 数据流的一部分的数据。 从后端发出的消息总是对应一个数据行，但是前端发出的消息可以任意分割数据流。

CopyDone (F & B)

Byte1('c')

标识这条消息是一个 COPY 结束指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

CopyFail (F)

Byte1('f')

标识这条消息是一个 `COPY` 失败指示器。

Int32

以字节记的消息内容的长度，包括长度本身。

String

一个报告失败原因的错误消息。

CopyInResponse (B)

Byte1('G')

标识这条消息是一条 StartCopyIn(开始拷贝输入)响应消息。 前端现在必须发送一条拷贝入数据(如果还没准备好做这些事情，那么发送一条 CopyFail 消息)。

Int32

以字节记的消息内容的长度，包括长度本身。

Int8

0 表示全部的 `COPY` 格式都是文本的(数据行由换行符分隔，字段由分隔字符分隔等等)。 1 表示都是二进制的(类似 DataRow 格式)。参阅COPY获取更多信息。

Int16

数据中要拷贝的字段数(由下面的 `_N_` 解释)。

Int16[`_N_`]

每个字段将要使用的格式代码，目前每个都必须是0(文本)或者1(二进制)。 如果全部拷贝格式都是文本的，那么所有的都必须是0。

CopyOutResponse (B)

Byte1('H')

标识这条消息是一条 StartCopyOut(开始拷贝输出)响应消息。这条消息后面将跟着一条拷贝出数据消息。

Int32

以字节记的消息内容的长度，包括它自己。

Int8

0 表示全部 `COPY` 格式都是文本(数据行由换行符分隔，字段由分隔字符分隔等等)。 1 表示所有拷贝格式都是二进制的(类似于 DataRow 格式)。参阅COPY获取更多信息。

Int16

要拷贝的数据的字段的数目(在下面的 `_N_` 说明)。

Int16[`_N_`]

每个字段要使用的格式代码。目前每个都必须是0(文本)或者1(二进制)。 如果全部的拷贝格式都是文本，那么所有的都必须是0。

CopyBothResponse (B)

Byte1('W')

标识这条消息是一条 StartCopyBoth(开始双向拷贝)响应消息。 这个消息只会被流复制使用。

Int32

以字节记的消息内容的长度，包括它自己。

Int8

0 表示全部拷贝 `COPY` 格式都是文本(数据行由换行符分隔，字段由分隔字符分隔等等)。 1 表示所有拷贝格式都是二进制的(类似于 DataRow 格式)。参阅COPY获取更多信息。

Int16

数据中要拷贝的字段数(由下面的 `_N_` 解释)。

Int16[`_N_`]

每个字段要使用的格式代码。目前每个都必须是0(文本)或者1(二进制)。 如果全部的拷贝格式都是文本，那么所有的都必须是0。

DataRow (B)

Byte1('D')

标识这个消息是一个数据行。

Int32

以字节记的消息内容的长度，包括长度自身。

Int16

后面跟着的字段值的个数(可能是零)。

然后，每个字段都会出现下面的数据域对：

Int32

字段值的长度，以字节记(这个长度不包括它自己)。 可以为0。一个特殊的情况是，-1 表示一个 NULL 的字段值。在 NULL 的情况下就没有跟着数据字段。

Byte `_n_`

一个字段的数值，以相关的格式代码表示的格式展现。 `_n_` 是上面的长度。

Describe (F)

Byte1('D')

标识消息是一个 Describe命令。

Int32

以字节记的消息内容的长度，包括字节本身。

Byte1

'`S`'描述一个预备语句；或者'`P`'描述一个入口。

String

要描述的预备语句或者入口的名字(或者一个空字符串，就会选取未命名的预备语句或者入口)。

EmptyQueryResponse (B)

Byte1('I')

标识这条消息是对一个空查询字符串的响应。 (这个消息替换了 CommandComplete。)

Int32(4)

以字节记的消息内容长度，包括它自己。

ErrorResponse (B)

Byte1('E')

标识消息是一条错误

Int32

以字节记的消息内容的长度，包括长度本身。

消息体由一个或多个标识出来的字段组成，后面跟着一个字节零作为终止符。字段可以以任何顺序出现。 对于每个字段都有下面的东西：

Byte1

一个标识字段类型的代码；如果为零，这就是消息终止符并且不会跟着有字符串。 目前定义的字段类型在Section 48.6列出。 因为将来可能增加更多的字段类型，所以前端应该不声不响地忽略不认识类型的字段。

String

字段值

Execute (F)

Byte1('E')

标识消息识一个 Execute 命令。

Int32

以字节记的消息内容的长度，包括长度自身。

String

要执行的入口的名字(空字符串选定未命名的入口)。

Int32

要返回的最大行数，如果入口包含返回行的查询(否则忽略)。零表示"没有限制"。

Flush (F)

Byte1('H')

标识消息识一条 Flush 命令。

Int32(4)

以字节记的消息内容的长度，包括长度本身。

FunctionCall (F)

Byte1('F')

标识消息是一个函数调用。

Int32

以字节记的消息内容的长度，包括长度本身。

Int32

声明待调用的函数的对象标识(OID)。

Int16

后面跟着的参数格式代码的数目(用下面的 `_c_` 表示)。 它可以是0，表示没有参数，或者是所有参数都使用缺省格式(文本)；或者是1，这种情况下声明的格式代码应用于所有参数；或者它可以等于参数的实际个数。

Int16[`_c_`]

参数格式代码。目前每个必须是0(文本)或者1(二进制)。

Int16

声明提供给函数的参数个数

然后，每个参数都出现下面字段对：

Int32

以字节记的参数值的长度(不包括长度自己)。可以为零。 一个特殊的例子是，-1 表示一个 NULL 参数值。如果是 NULL ，则没有参数字节跟在后面。

Byte `_n_`

参数的值，格式是用相关的格式代码表示的。 `_n_` 是上面的长度。

在最后一个参数之后，出现下面的字段：

Int16

函数结果的格式代码。目前必须是0(文本)或者1(二进制)。

FunctionCallResponse (B)

Byte1('V')

标识这条消息是一个函数调用结果

Int32

以字节计的消息内容长度，包括长度本身。

Int32

以字节记的函数结果值的长度(不包括长度本身)。 可以为零。一个特殊的情况是 -1 表示 NULL 函数结果。如果是 NULL 则后面没有数值字节跟随。

Byte `_n_`

函数结果的值，格式是相关联的格式代码标识的。 `_n_` 是上面的长度。

NoData (B)

Byte1('n')

标识这条消息是一个无数据指示器

Int32(4)

以字节计的消息内容长度，包括长度本身。

NoticeResponse (B)

Byte1('N')

标识这条消息是一个通知

Int32

以字节计的消息内容长度，包括长度本身。

消息体由一个或多个标识字段组成，后面跟着字节零作为中止符。字段可以以任何顺序出现。对于每个字段，都有下面的东西：

Byte1

一个标识字段类型的代码；如果为零，那么它就是消息终止符，并且后面不会跟着字符串。目前定义的字段类型在Section 48.6里列出。 因为将来可能会增加更多字段类型，所以前端应该将不识别的字段安静地忽略掉。

String

字段值

NotificationResponse (B)

Byte1('A')

标识这条消息是一个通知响应

Int32

以字节计的消息内容长度，包括长度本身。

Int32

通知后端进程的进程 ID

String

触发通知的通道(channel)的名字

String

从通知进程传递过来的"装载（payload）"字符串。

ParameterDescription (B)

Byte1('t')

标识消息是一个参数描述

Int32

以字节计的消息内容长度，包括长度本身。

Int16

语句所使用的参数的个数(可以为零)

然后，对每个参数，有下面的东西：

Int32

声明参数数据类型的对象 ID

ParameterStatus (B)

Byte1('S')

标识这条消息是一个运行时参数状态报告

Int32

以字节计的消息内容长度，包括长度本身。

String

被报告的运行时参数的名字

String

参数的当前值

Parse (F)

Byte1('P')

标识消息是一条 Parse 命令。

Int32

以字节计的消息内容长度，包括长度本身。

String

目的预备语句的名字(空字符串表示选取了未命名的预备语句)

String

要解析的查询字符串

Int16

声明的参数数据类型的数目(可以为零)。 请注意这个参数并不意味着可能在查询字符串里出现的参数个数的意思，只是前端希望预先声明的类型的数目。

然后，对每个参数，有下面的东西：

Int32

声明参数数据类型的对象 ID 。在这里放一个零等效于不声明该类型。

ParseComplete (B)

Byte1('1')

标识这条消息是一个 Parse 完成指示器。

Int32(4)

以字节计的消息内容长度，包括长度本身。

PasswordMessage (F)

Byte1('p')

标识这条消息是一个口令响应。 注意这也被使用在GSSAPI和SSPI响应消息中。 （这实际上是一个设计错误，既然这里包含的数据不是一个NULL终止的字符串，而可能是任意的二进制数据。）

Int32

以字节计的消息内容长度，包括长度本身。

String

口令(必要的情况下，是加密后的)。

PortalSuspended (B)

Byte1('s')

标识这条消息是一个入口挂起指示器。 请注意这个消息只出现在达到一条 Execute 消息的行计数限制的时候。

Int32(4)

以字节计的消息内容长度，包括长度本身。

Query (F)

Byte1('Q')

标识消息是一个简单查询。

Int32

以字节计的消息内容长度，包括长度本身。

String

查询字符串自身

ReadyForQuery (B)

Byte1('Z')

标识消息类型。在后端为新的查询循环做好准备的时候，总会发送 ReadyForQuery

Int32(5)

以字节计的消息内容长度，包括长度本身。

Byte1

当前后端事务状态指示器。 可能的值是空闲状况下的'`I`'(不在事务块里)；在事务块里是'`T`'；或者在一个失败的事务块里是'`E`'(在事务块结束之前，任何查询都将被拒绝)。

RowDescription (B)

Byte1('T')

标识消息是一个行描述

Int32

以字节计的消息内容长度，包括长度本身。

Int16

声明在一个行里面的字段数目(可以为零)

然后对于每个字段，有下面的东西：

String

字段名字

Int32

如果字段可以标识为一个特定表的字段，那么就是表的对象 ID ；否则就是零。

Int16

如果该字段可以标识为一个特定表的字段，那么就是该表字段的属性号；否则就是零。

Int32

字段数据类型的对象 ID

Int16

数据类型尺寸(参阅 `pg_type.typlen`)。请注意负数表示变宽类型。

Int32

类型修饰词(参阅 `pg_attribute.atttypmod`)。修饰词的含义是类型相关的。

Int16

用于该字段的格式码。目前会是0(文本)或者1(二进制)。 从语句的Describe 返回的 RowDescription 里，格式码还是未知的，因此总是零。

SSLRequest (F)

Int32(8)

以字节计的消息内容长度，包括长度本身。

Int32(80877103)

SSL请求码。 选取的数值在高 16 位里包含 `1234`，在低 16 位里包含 `5679`。为了避免混淆，这个编码必须和任何协议版本号不同。

StartupMessage (F)

Int32

以字节计的消息内容长度，包括长度本身。

Int32(196608)

协议版本号。高 16 位是主版本号(对这里描述的协议而言是 3)。低 16 位是次版本号(对于这里描述的协议而言是 0)。

协议版本号后面跟着一个或多个参数名和值字符串的配对。 要求在最后一个名字/数值对后面有个字节零。参数可以以任意顺序出现。 `user` 是必须的，其它都是可选的。每个参数都是这样声明的：

String

参数名。目前可以识别的名字是：

`user`

用于连接的数据库用户名。必须；无缺省。

`database`

要连接的数据库。缺省是用户名。

`options`

给后端的命令行参数。(这个特性已经废弃，更好的方法是设置单独的运行时参数。)

除了上面的外，在后端启动的时候可以设置的任何运行时参数都可以列出来。 这样的设置将在后端启动的时候附加(在分析了命令行参数之后，如果有的话)。这些值将成为会话缺省。

String

参数值

Sync (F)

Byte1('S')

表示该消息为一条 Sync 命令。

Int32(4)

以字节计的消息内容长度，包括长度本身。

Terminate (F)

Byte1('X')

标识消息是一个终止消息。

Int32(4)

以字节计的消息内容长度，包括长度本身。

48.6. 错误和通知消息字段

本节描述那些可能出现在 ErrorResponse 和 NoticeResponse 消息里的字段。 每个字段类型有一个单字节标识记号。请注意，任意给定的字段类型在每条消息里都应该最多出现一次。

`S`

严重性：该字段的内容是 `ERROR`，`FATAL`，`PANIC`(在一个错误消息里)，`WARNING`，`NOTICE`，`DEBUG`，`INFO`，`LOG`(在一条通知消息里)之一，或者是这些的某种本地化翻译的字符串。总是会出现。

`C`

代码：错误的 SQLSTATE 代码(参阅Appendix A)。不能本地化。总是出现。

`M`

消息：人类可读的错误消息的主体。这些消息应该准确并且简洁(通常是一行)。总是出现。

`D`

细节：一个可选的从属错误消息，承载有关问题的更多错误消息。可以是多行。

`H`

提示：一个可选的有关如何处理问题的建议。它和细节不同的地方是它提出了建议(可能并不合适)而不仅仅是事实。可以是多行。

`P`

位置：这个字段值是一个十进制 ASCII 整数，表示一个错误游标的位置，它是一个指向原始查询字符串的索引。 第一个字符的索引是 1， 位置是以字符计算而非字节计算的。

`p`

内部位置：这个域和 `P` 域定义相同，但是它用于游标的位置指向一个内部生成的命令，而不是用于客户端提交的命令。这个字段出现的时候，总是会出现 `q` 字段。

`q`

内部查询：失败的内部生成的命令的文本。比如，它可能是一个 PL/pgSQL 函数发出的 SQL 查询。

`W`

哪里：一个指示错误发生的环境的指示器。目前，这个参数包含一个活跃的过程语言函数的调用堆栈的追溯和内部生成的查询。这个追溯每条记录一行，最新的在最上面。

`s`

模式名：如果错误和一个特定的数据库对象相关，则是包含该对象的模式的名字。

`t`

表名：如果错误和一个特定的表相关，则是这个表的名字。（表的模式名参考模式名字段）

`c`

列名:如果错误和一个特定的列相关，则是这个列的名字。（特定的表参考模式名和表名字段）

`d`

数据类型名:如果错误和一个特定的数据类型相关，则是这个数据类型的名字。(数据类型的模式名参考模式名字段)

`n`

约束名:如果错误和一个特定的约束相关，则是这个约束的名字。相关的表或者域参考上面列出的字段。（因为这个目的，索引被当作约束对待，即使它们不是通过约束语法创建的）

`F`

文件：报告错误的源代码位置的文件名。

`L`

行：报告错误的源代码位置的行号。

`R`

过程：报告错误的源代码过程名。

> **Note:** 模式名、表名、列名、数据类型名以及约束名相关的字段只是提供给有限的错误类型。参考Appendix A。前端不应该假设只要出现了其中一个字段就一定会有其它字段。核心的错误源遵守上面提到的相互关系，但是用户自定义函数可能以不同的方式使用这些字段。同样，客户端不应该假设这些字段指示了在当前数据库中同时存在的对象。

客户端负责对显示信息进行格式化输出以符合需要；特别是它应该根据需要断开长的行。在错误消息字段里出现的换行符应该当作一个分段的符号，而不是换行。

48.7. 自协议 2.0 以来的变化的概述

本节提供一个快速的变化检查列表，以便于那些试图将现有的客户端库更新到 3.0 协议的开发人员。

初始化的启动包用了一个灵活的字符串列表格式取代了固定的格式。 请注意，运行时参数的会话缺省值现在可以直接在启动包中声明。 (实际上，你可以在使用 `options` 字段之前干这件事情，但是因为 `options` 的宽度限制以及缺乏引用值中的空白的方法，这并不是很安全的技巧。)

现在所有的消息在消息类型字节后面都有一个长度计数(除了启动包之外，它没有类型字节)。同时还要注意现在 PasswordMessage 有一个类型字节。

ErrorResponse 和 NoticeResponse(' `E` '和' `N` ')消息现在包含多个字段，从这些字段里客户端代码可以组合出自己所希望的详细程度的错误消息。 请注意独立的字段通常不是用换行符终止的，虽然在老协议里发送的单个字符串总是会用换行符终止。

The ReadyForQuery(' `Z` ')消息包括一个事务状态指示符。

BinaryRow 和 DataRow 消息类型之间的区别不再存在了；单个 DataRow 消息类型用于返回所有格式的数据。 请注意 DataRow 的布局已经改变成比较容易分析了。同样，二进制数值的表现形式已经改变了：它不再是直接和服务器的内部表现形式绑定。

有了一种新的"扩展查询"的子协议，它增加了前端消息类型 Parse, Bind, Execute, Describe, Close, Flush, Sync ，以及后端消息类型 ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData, CloseComplete 。 现有的客户端不用关心这个子协议，但是利用这个子协议将可能改进性能或者功能。

`COPY` 数据现在封装到了 CopyData 和 CopyDone 消息里。 现在有种很好的方法从正在进行的 `COPY` 动作中恢复错误。 最后一行的特殊的" `\.` "不再必须了，并且在 `COPY OUT` 的过程中不再发送。 （在 `COPY IN` 的时候它仍然被认为是一个终止符，但是它的使用已经废弃了并且最终将被删除。） 现在支持二进制 `COPY` 。CopyInResponse 和 CopyOutResponse 消息包括指示字段数目和每个字段格式的信息域。

FunctionCall 和 FunctionCallResponse 消息的布局变化了。 FunctionCall 现在支持给函数传递 NULL 参数。 它同样可以处理以文本或者二进制格式传递参数和检索结果。 不用再认为 FunctionCall 有潜在的安全性漏洞，因为它并不提供对内部服务器数据表现形式的直接访问。

后端在启动的时候为它认为客户端库感兴趣的所有参数发送 ParameterStatus(' `S` ')消息。 随后，如果这些参数的活跃值发生变化，那么发送一条 ParameterStatus 消息。

RowDescription(' `T` ')消息为所描述的行的每个字段装载新表的 OID 和字段号数据域。它同样还为每个字段显示了格式代码。

后端不再生成 CursorResponse(' `P` ')消息。

NotificationResponse(' `A` ')消息有一个额外的字符串域，它可以携带来自 `NOTIFY` 事件发送者的"装载（payload）"数据。

EmptyQueryResponse(' `I` ')以前包含一个空字符串参数；这个已经被删除了。

Chapter 49. PostgreSQL 编码约定

Table of Contents

49.1. 格式

代码格式使用每个制表符 4 列的空白，目前是保留制表符状态 (也就是说制表符不被展开为空白)。每个逻辑缩进层次都是更多的一个制表符。

布局规则(花括弧定位等)遵循 BSD 传统。特别的， `if` 、 `while` 、 `switch` 等的控制块的大括号从它们自己的行开始。

限制行的长度，这样代码在80字段的窗口内可读。（这并不意味着你不能超过80个字段。 例如，在任意地方分解一个长的错误消息字符串，只是为了保持代码在80个字段内， 可能不是为了可读性。）

不要使用C++风格的注释(`//` 注释)。严格的ANSI C编译器不接受它们。 相同的原因，不要使用C++扩展，比如在块中声明新的变量。

多行注释块的首选风格是

```
/*
 * comment text begins here
 * and continues here
 */
```

请注意，在第一列开始的注释块将被pgindent按原样保存， 但是它将回流注释块缩进，就像它们是纯文本。如果你想要在缩进块中保留换行， 像这样添加破折号：

```
    /*----------
     * comment text begins here
     * and continues here
     *----------
     */
```

虽然提交的补丁并不需要完全遵循这些格式化规则，最好还是这么做。 你的代码将会在下一个版本之前被pgindent处理， 虽然这样做不会让它看起来比其它的格式化习惯更好看。 一个补丁的好的经验法则是"让新的代码看起来像是被已有的代码包围"。

`src/tools` 目录包含了适用于emacs 的示范配置文件，xemacs或vim 用户也必须确保其格式代码也符合上述规范。

文本浏览工具more和less 可以用下面命令调用：

```
more -x4
less -x4
```

这样就可以让他们正确显示制表符。

49.2. 报告服务器里的错误

在服务器代码里生成的错误、警告以及日志信息应该用 `ereport` 或者它的兄弟 `elog` 创建。这个函数的使用已经复杂得需要做些解释了。

每条消息都有两个必须的要素：一个严重级别(范围从 `DEBUG` 到 `PANIC`)和一个主要消息文本。除此之外还有可选的元素， 最常见的就是一个遵循 SQL 标准的 SQLSTATE 习惯的错误标识码。 `ereport` 本身只是一个壳函数， 它的存在主要是为了便于让消息生成看起来像 C 代码里的函数调用。 `ereport` 直接接受的唯一参数是严重级别。 主消息文本和任何附加消息元素都是通过在 `ereport` 调用里调用辅助函数(比如 `errmsg`)生成的。

典型的调用 `ereport` 的方式看起来可能像下面这样：

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

这样就声明了错误严重级别 `ERROR` (一个普通错误)。 `errcode` 调用指定一个在 `src/include/utils/errcodes.h` 里面使用宏定义的SQLSTATE错误代码。 `errmsg` 调用提供主要的消息文本。 请注意包围在辅助函数调用周围的额外的圆括弧—这么做虽然烦人， 但是语法上是必须的。

然后是一个更复杂的例子：

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.")));
```

这个例子演示了使用格式化代码把运行时数值嵌入一个消息文本的用法。 同样，还提供了一个可选的"暗示"信息。

`ereport` 可用的附属过程有：

- `errcode(sqlerrcode)` 为该条件声明 SQLSTATE 错误标识符代码。 如果没有调用这个过程，并且错误严重级别是 `ERROR` 或更高， 那么错误标识符缺省是 `ERRCODE_INTERNAL_ERROR` ， 如果错误严重级别是 `WARNING` 则为 `ERRCODE_WARNING` ， 否则(用于 `NOTICE` 或者更低级别)为 `ERRCODE_SUCCESSFUL_COMPLETION` 。 虽然这些缺省都很方便，但是最好还是在省略 `errcode()` 调用之前三思。

- `errmsg(const char *msg, ...)` 声明主错误消息文本， 以及可能的插入其中的运行时数值。插入是使用 `sprintf` 风格的格式代码插入的。除了 `sprintf` 接受的标准格式代码，还接受 `%m` 用于插入 `strerror` 为当前 `errno` 值返回的错误信息。 [1] `%m` 并不要求在 `errmsg` 的参数列表里有任何对应的项目。 请注意这个消息字符串在格式代码得到处理之前将不会通过 `gettext` 运行获取合适的本地化。

- `errmsg_internal(const char *msg, ...)` 和 `errmsg` 一样，只是消息字符串将不会包含在国际化消息字典里。这个函数应该用于 "不可能发生"的情况，也就是不值得展开进行翻译的场合。

- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` 就像是 `errmsg` ， 但是支持消息的各种复数形式。 `_fmt_singular_` 是英语单数形式， `_fmt_plural_` 是英语复数形式， `_n_` 是决定使用哪个复数形式的整数值，剩余的参数根据选择的格式字符串进行格式化。 更多信息请查看Section 50.2.2。

- `errdetail(const char *msg, ...)` 提供一个可选的"详细"信息； 在存在额外的信息，并且很适合放在主消息里面的时候使用这个函数。 消息字符串处理的方法和 `errmsg` 完全一样。

- `errdetail_internal(const char *msg, ...)` 和 `errdetail` 一样， 只是消息字符串将不会包含在国际化消息字典里。这个函数应该用于不值得展开进行翻译的详细消息， 比如，因为它们太技术了以至于对大多数用户来说无用。

- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ..` 类似于 `errdetail` ， 但是支持消息的各种复数形式。更多信息请查阅Section 50.2.2。

- `errdetail_log(const char *msg, ...)` 和 `errdetail` 一样，除了这个字符串只到服务器日志，从不到客户端。 如果 `errdetail` （或它的以上相同的其中之一）和 `errdetail_log` 都使用了，那么一个字符串到客户端，另一个到服务器日志。 这对于过于安全敏感或太大而不能包含进发送到客户端的报告中的错误细节来说是有用的。

- `errhint(const char *msg, ...)` 提供一个可选的"暗示"消息； 这个函数用于提供如何修补问题的建议，而不是提供错误的事实。 消息字符串处理的方式和 `errmsg` 一样。

- `errcontext(const char *msg, ...)` 通常不会直接从 `ereport` 消息点里直接调用；而是在 `error_context_stack` 回调函数里提供有关错误发生的环境的信息，比如，当前的位置是在一个 PL 函数里等等。 消息字符串的处理和 `errmsg` 完全一样。和其它辅助函数不同， 这个函数在一次 `ereport` 调用里可以调用多次； 随后的调用生成的字符串将带着各自的换行符连接在原来的字符串上。

- `errposition(int cursorpos)` 声明一个错误在查询字符串里的文本位置。 目前它只是在汇报查询处理过程中的词法和语法分析阶段检测到的错误有用。

- `errtable(Relation rel)` 指定一个关系， 关系名和模式名应该作为辅助字段包含在错误报告内。

- `errtablecol(Relation rel, int attnum)` 指定一个字段， 该字段的名字、表名和模式名应该作为辅助字段包含在错误报告中。

- `errtableconstraint(Relation rel, const char *conname)` 指定一个表约束，该表约束的名字、表名和模式名应该作为辅助字段包含在错误报告中。 不管它们是否有一个相关的 `pg_constraint` 项，索引都应该被认为是约束。 小心地传递底层堆关系，不只是索引本身，作为 `rel` 。

- `errdatatype(Oid datatypeOid)` 指定一个数据类型， 它的名字和模式名应该作为辅助字段包含在错误报告中。

- `errdomainconstraint(Oid datatypeOid, const char *conname)` 指定一个域约束，该域约束的名字、域名和模式名应该作为附属字段包含在错误报告中。

- `errcode_for_file_access()` 是一个便利函数， 它可以为一个文件访问类的系统调用选择一个合适的 SQLSTATE 错误标识符。 它利用保存下来的 `errno` 判断生成哪个错误代码。 通常它应该和主错误消息文本里的 `%m` 结合使用。

- `errcode_for_socket_access()` 是一个便利函数， 它可以为一个套接字相关的系统调用选择一个合适的 SQLSTATE 错误标识符。

- `errhidestmt(bool hide_stmt)` 可以用来在主日志中指定消息的 `STATEMENT:` 部分的消除。 如果消息文本早已包括当前语句， 那么这通常是合适的。

> **Note:** `errtable` 、 `errtablecol` 、 `errtableconstraint` 、 `errdatatype` 或 `errdomainconstraint` 中最多只有一个函数应该用在一个 `ereport` 调用中。 这些函数的存在允许应用提取与没有检测潜在本地化错误消息文本的错误条件相关的数据库对象的名字。 这些函数应该用在应用希望自动错误处理的错误报告中。自PostgreSQL 9.3起， 完全覆盖只是为了SQLSTATE class 23中的错误而存在（违反完整约束）， 但是有可能会在未来扩展。

还有一个老一些的 `elog` 函数，仍然在频繁使用。一个 `elog` 调用：

```
elog(level, "format string", ...);
```

完全等效于：

```
ereport(level, (errmsg_internal("format string", ...)));
```

请注意 SQLSTATE 错误代码总是缺省的，并且消息字符串并没有翻译。因此， `elog` 应该只用于内部错误以及低层的调试日志。 任何普通用户感兴趣的消息都应该通过 `ereport` 生成。当然， 还有大量内部的"不可能发生"的错误检查使用 `elog` ； 因为这些信息最好还是表示得简单些好。

书写好的错误消息的建议可以在Section 49.3找到。

Notes

[1] 也就是说，在到达 `ereport` 调用的时候当前的数值； 在附属报告过程里对 `errno` 的修改将不会影响他。但是如果你在 `errmsg` 的参数列表里明确地写 `strerror(errno)`， 这一点就不能保证了，因此，请不要这么做。

49.3. 错误消息风格指导

这份风格向导的目的是希望能把所有PostgreSQL 生成的消息维护一个一致的，用户友好的风格。

49.3.1. 何去何从

主信息应该简短，基于事实，并且避免引用类似特定函数名等这样的实现细节。 "简短"意味着"在正常情况下应该能放在一行里"。 如果需要保持主信息的简短，或者如果你觉得需要提到失败的特定系统调用之类的实现细节， 可以使用一个详细信息。主信息和详细信息都应该是基于事实的。 使用一个提示消息给出一个修补问题的提示，特别是在提出的建议可能并不总是有效的情况下。

比如，可以不这么写：

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

而是：

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

基本原理：保持主消息的简短可以使它的内容有效， 并且让客户端的屏幕空间布局可以做出给错误信息保留一行就足够的假设。 而详细信息和提示信息可以转移到一个冗余模式里，或者使一个弹出的错误细节的窗口。 同样，详细信息和提示信息通常都会在服务器日志里消除，以节约空间。 对实现细节的引用最好避免，因为毕竟用户不知道细节。

49.3.2. 格式

不要在消息文本里放任何有关格式化的特定的假设。 除非是客户端或者服务器日志为了符合自己需要回卷了长行。在长信息里， 新行字符(\n)可以用于分段建议。不要用新行结束一条消息。 不要使用 tab 或者其它格式化字符。在错误环境下的显示里， 系统会自动给独立级别的环境，比如，函数调用，增加新行。

基本原理：信息不一定非得在终端类型的显示器上显示。 在 GUI 显示或者在浏览器里，这些格式指示器最好被忽略。

49.3.3. 引号

在需要的时候，英文文本应该使用双引号引起来。其它语言的文本应该一致地使用一种引号， 这种用法应该和出版习惯以及其它程序的计算输出一致。

基本原理：选择双引号而不是单引号从某种角度来说是随机选择，但是应该是最优的选择。有人建议过根据 SQL 传统，在不同对象类型上使用不同的引号(也就是说， 字符串单引号，标识符双引号)。但是这是一种语言内部的技巧，许多用户甚至都不熟悉， 并且也不能厌战到其它类型的引号场合，也不能翻译成其它语言，而且也没啥意义。

49.3.4. 使用引号

总是用引号分隔文件名，用户提供的标识符，以及其它可能包含字的变量。 不要用引号包含那些不会包含字的变量(比如，操作符)。

在后端里有些函数会根据需要在他们的输出上加双引号 (比如 `format_type_be()`)。不要在这类函数的输出上加额外的引号。

基本原理：对象的名字嵌入到信息里面之后，可能造成歧义。 在一个插入的名字的起始和终止的位置保持一致。 但是不要在信息里混杂大量不必要的或者重复的引号。

49.3.5. 语法和标点

对于主错误信息和详细/提示信息，规则不同：

主错误信息：首字母不要大写。不要用句号结束信息。绝对不要用叹号结束一条信息。

详细和提示信息：使用完整的句子，并且用句号终止每个语句。句子首字母大写。 如有后面有另外一个句子，那么在句号后面放置两个空格（对于英文来说； 可能不适合其他语言）。

错误上下文字符串：不要大写首字母，不要用句号结束字符串。 上下文字符串应该通常不是完整的句子。

基本原理：避免标点可以让客户端应用比较容易的把信息嵌入到各种语言环境中。并且， 主消息也经常不是完整的句子。并且，如果信息长得超过一个句子， 那么就应该把他们分裂成主信息和详细信息部分。不过， 详细和提示信息长得多并且可能需要包含多个句子。为了保持一致， 这些句子应该遵循完整句子得风格，即使他们只有一个句子。

49.3.6. 大写字符与小写字符比较

消息用语使用小写字符，包括主错误信息的首字母。 如果消息中出现 SQL 命令和关键字，用大写。

基本原理：这样很容易让所有东西看起来都一样，因为有些消息是完整的句子，有些不是。

49.3.7. 避免被动语气

使用主动语气。如果有主语，那么就使用完整的句子("A不能做 B")。 如果主语是程序自己，那么就使用电报风格的语言；不要用"我"作为程序的主语。

基本原理：程序不是人。不要假装成其他的。

49.3.8. 现代时与过去时的比较

如果尝试某事失败，但可能下次尝试的时候成功(可能是修补了某些问题之后)， 那么使用过去时。如果错误肯定是永久的，那么用现代时。

下面的两个形式的句子之间有重要的语义差别：

```
could not open file "%s": %m
```

和：

```
cannot open file "%s"
```

第一个句子的意思是打开某个文件的企图失败。这个信息应该给出一个原因， 比如说"磁盘满"或者"文件不存在"之类的。 过去时的语气应该是合适的，因为下次磁盘可能不再是满的，或者有问题的文件存在了。

第二种形式表示打开指定文件的功能根本就不在程序里存在，或者是这么做概念上是错误的。 现代时语气是合适的，因为这个条件将无限期存在。

基本原理：当然，普通用户将不会仅仅从信息的时态上得出大量的结论， 但是既然语言提供给语法，那么就应该正确使用。

49.3.9. 对象类型

在引用一个对象的名字的时候，说明它是什么类型的对象。

基本原理：否则，没人知道"foo.bar.baz"指的是什么。

49.3.10. 方括弧

方括弧只用(1)在命令概要里表示可选的参数，或者(2)表示一个数组下标。

基本原理：任何其它的东西都不能对应这两种众所周知的习惯用法并且会让人混淆。

49.3.11. 组装错误信息

如果一个信息包含其它地方生成的文本，用下面的风格包含它：

```
could not open file %s: %m
```

基本原理：很难估计所有可能放在这里的错误代码并且把它放在一个平滑的句子里， 所以需要某种方式的标点。也曾经建议把嵌入的文本放在圆括弧里， 但是如果嵌入文本是信息的最重要部分，那么就不太自然，而这种情况是很经常的。

49.3.12. 错误的原因

消息应该总是说明为什么发生错误。比如：

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

如果不知道原因，那么你最好修补代码。

49.3.13. 函数名

不要在错误信息里包含报告过程的名字。需要的时候，有别的机制找出这个函数，并且， 对于大多数用户，这个信息也没什么用。如果错误信息在缺少函数名的情况下没有什么意义，那么重新措辞。

```
BAD:    pg_atoi: error in "z": cannot parse "z"
BETTER: invalid input syntax for integer: "z"
```

也避免提及被调用的函数名字；应该说代码试图做什么：

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

如果确实必要，在详细信息里提出系统调用。在某些场合下， 提供给系统调用的具体数值是适合放在详细信息里的。

基本原理：用户不知道这些函数都干些啥。

49.3.14. 尽量避免的字眼

Unable/不能**.** "Unable/不能" 几乎是被动语气。最好使用 "cannot/无法"或者"could not"。

Bad/坏的**.** 类似"bad result/坏结果"这样的错误信息真的是很难聪明地解释。 最好写出为什么结果是"bad/坏的"，比如，"invalid format/非法格式"。

Illegal/非法**.** "Illegal/非法"表示违反了法律，剩下的就是"invalid/无效的"。 但是最好还是说为什么无效。

Unknown/未知**.** 尽量避免使用"unknown/未知"。想想"error: unknown response"。 如果你不知道响应是什么，你怎么知道是错误?"Unrecognized/无法识别的" 通常是更好的选择。还有最好要包括被抱怨的数值。

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find/找到 和 **Exists/**存在 比较**.** 如果程序使用一个相当复杂的算法来定位一个资源(比如，一个路径搜索)， 并且算法失败了，那么说程序无法"find/找到"该资源是合理的。 但是，如果预期的资源位置是已知的但是程序无法在那里访问它， 那么说这个资源不"exist/存在"。 这种情况下用"find/找到"听起来语气比较弱并且会混淆事实。

May 和 **Can** 和 **Might** 比较**.** "May"暗示权限（如，"You may borrow my rake."）， 在文档或错误消息中很少使用。"Can"暗示能力 （如，"I can lift that log."），"might"暗示可能性（如，"It might rain today."）使用适当的单词澄清意义和帮助翻译。

缩写**.** 避免缩写，像"can't"；使用"cannot"代替。

49.3.15. 正确地拼写

用单词的全拼。比如，避免下面这样的缩写：

- spec
- stats
- parens
- auth
- xact

基本原理：这样将改善一致性。

49.3.16. 本地化

请记住，错误信息文本是需要翻译成其它语言的。遵循Section 50.2.2 里面的指导以避免给翻译者造成太多麻烦。

Chapter 50. 本地语言支持

Table of Contents

50.1. 寄语翻译家

PostgreSQL 程序(服务器和客户端)可以用你喜爱的语言提示消息，只要那些消息被翻译过。创建和维护翻译过的消息集需要那些熟悉自己的语言并且希望为 PostgreSQL 做一些事情的人。 实际上要干这件事你完全不必是个程序员。本章讲的就是如何帮助 PostgreSQL 做这件事。

50.1.1. 要求

本章是有关软件工具的，不会评判你的语言能力。理论上，你只需要一个文本编辑器。 但这种情况只存在于你不想看看自己翻译的消息的情况下，这种情况很难发生吧。 在你配置你的源程序的时候，记住要使用 `--enable-nls` 选项。 这样也会检查 libintl 库和 `msgfmt` 程序，它们是所有最终用户都需要的东西。 要试验你的工作，遵照安装指导中相应的部分就可以了。

如果你想开始一个新的翻译工作或者想做消息表合并工作(下面描述)，那么你就还分别需要有 GNU 兼容的 `xgettext` 和 `msgmerge` 。 稍后将试着整理它，这样，如果你使用的是一个打包的源码发布，那么你就不再需 要 `xgettext` (从 Git 里下源码还是需要的)。现在推荐 GNU Gettext 0.10.36 或者更高版本。

你的本地 gettext 实现应该和它自身的文档在一起发布。 其中有一些可能和下面的内容重复，但如果要知道额外的细节，你应该看看它们。

50.1.2. 概念

消息原文(英语)和它们的(可能)翻译过的等价物都放在消息表 里，每个程序一个(不过相关的程序可以共享一个消息表)以及每种目标语言一个。 消息表有两种文件格式：第一种是"PO"(可移植对象)文件，它是纯文本文件， 带一些翻译者编辑的特殊语法。第二种是"MO"(机器对象)文件， 它是从相应的 PO 文件生成的二进制文件，在国际化了的程序运行的时候使用。 翻译者并不处理 MO 文件；实际上几乎没人处理它。

消息表的文件扩展名分别是 `.po` 或 `.mo` 就一点也不奇怪了。主文件名要么是它对应的程序名，要么是该文件适用的语言， 视情况而定。这种做法有点让人混淆。比如 `psql.po` (psql 的 PO 文件)或者 `fr.mo` (法语的 MO 文件)。

PO 文件的文件格式如下所示：

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

msgid 是从程序源代码中抽取的(其实不必是从源码，但这是最常用的方法)。 msgstr 行初始为空，由翻译者填充有用的字符串。 该字符串可以包含 C 风格的逃逸字符并且可以像演示的那样跨行继续。 下一行必须从该行的开头开始。

字符引入一个注释。如果 # 后面紧跟着空白，那么这是翻译者维护的注释。 文件里也可能有自动注释，它们是在 # 后面紧跟着非空白字符。 这些是由各种在 PO 文件上操作的工具生成的，主要目的是帮助翻译者。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

.风格的注释是从使用消息的源文件中抽取的。 程序员可能已经插入了一些消息给翻译者，比如那些预期的格式等。 #: 注释表示该消息在源程序中使用的准确位置。 翻译者不需要查看程序源文件，不过如果他觉得翻译得不对劲，那么他也可以查看。 #, 注释包含从某种程度上描述消息的标志。 目前有两个标志：如果该消息因为程序源文件的修改变得过时了，那么设置 `fuzzy` (模糊)标志。 翻译者然后就可以核实这些，然后可能删除这个fuzzy标志。请注意fuzzy消息是最终用户不可见的。 另外一个标志 `c-format` ,表示该消息是一个 `printf` 风格的格式模版。 这就意味着翻译也应该是一个格式化字

符串，带有相同数目和相同类型的占位符。 有用于核实这些的工具，它们生成 c 风格标志的键。

50.1.3. 创建和维护消息表

好，那怎么创建一"空白"的消息表呢? 首先，进入包含你想翻译消息的程序所在的目录。 如果那里有一个文件叫 `nls.mk` ，那么这个程序就已经准备好翻译了。

如果目录里已经有一些 `.po` 文件了，那么就是有人已经做了一些翻译工作了。 这些文件是用 `_language_`.po 命名的， 这里的 `_language_` 是ISO 639-1 规定的两字母语言代码(小写)，比如 `fr.po` 指的是法语。 如果每种语言需要多过一种的翻译，那么这些文件也可以叫做 `_language_` `_*region*`.po 这里的 *region* 是 [ISO 3166-1 规定的两字母国家代码(大写)](http://www.iso pt_BR.po` 指的是巴西葡萄牙语。如果你找到了你想要的语言文件，那么你就可以在那个文件上干活。

如果你需要开始一个新的翻译工作，那么首先运行下面的命令

```
gmake init-po
```

这样将创建一个文件 `_progname_`.pot。 (用 `.pot` 和"生产中"使用的 PO 文件区分开。 `T` 代表"template"。) 把这个文件拷贝成 `_language_`.po 然后编辑它。要让程序知道有新语言可以用，还要编辑文件 `nls.mk` 以增加该语言(或者语言和国家)代码到类似下面这样的行：

```
AVAIL_LANGUAGES := de fr
```

（当然可能还有其它国家。）

随着下层的程序或者库的改变，程序员可能修改或者增加消息。 这个时候，你不必从头再来。只需要运行下面的命令

```
gmake update-po
```

它将创建一个新的空消息表文件(你开始时用的 pot 文件)并且会把它和现有的 PO 文件合并起来。 如果合并算法不能确定某条具体的消息是否合并正确，那么它就会像上面解释那样把它定义为"fuzzy"(模糊)。 新的PO文件会保存为带 `.po.new` 扩展的文件。

50.1.4. 编辑 PO 文件

PO 文件可以用普通的文本编辑器编辑。 翻译者应该只修改那些在 msgstr 指示符后面的双引号中间的内容，也可以增加评注和修改模糊(fuzzy)标志。 Emacs 有一个用于 PO 的模式，我觉得相当好用。

不需要把 PO 文件完全填满。如果有些字符串没有翻译(或者是一个空白的翻译)，那么软件会自动使用原始的字符串。 也就是说提交一个未完成的翻译包括在源码树中并不是一个问题；那样可以留下让其他人继续你的工作的空间。 不过，鼓励你在完成一次合并之后首先消除模糊的条目。 要知道模糊的条目是不会被安装的；它们只起到表示可能是正确翻译的引用的作用。

下面是一些编辑翻译消息的时候要记住的事情：

- 确保如果原始消息是以换行结尾的话，翻译后的消息也如此。类似的情况适用于 tab 等。
- 如果最初的字符串是 `printf` 格式的字符串，那么翻译串也必须如此。 翻译串还要有同样的格式声明词，并且顺序相同。 有时候语言的自然规则会让这么做几乎不可能或者及其难看。这时候你可以用下面的格式：

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  这样，第一个占位符实际上使用列表里的第二个参数。 `_digits_` $ 应该跟在 % 后面并且在其它格式操作符之前。 这个特性实际上存在于 `printf` 函数族中。 你可能从来没有听说过它，因为除了消息国际化以外它没有什么用处。
- 如果原始的字符串包含语言错误，那么请报告它(或者在程序源文件中直接修补它)，然后按照正常翻译。 正确的字符串可能在程序源文件更新的时候被合并进来。 如果最初的字符串与事实不符，那么请报告它(或者自己修补它)但是不要翻译它。 相反，你可以在 PO 文件中用注解给该字符串做个标记。
- 维护风格和原始字符串的语气。特别是那些不成句子的消息(`cannot open file %s`) 可能不应该以大写字符开头(如果你的语言区分大小写的话)或者用句号结束(如果你的语言里有符号标志)。 阅读节Section 49.3可能会有所帮助。
- 如果你不知道一条消息是什么意思，或者它很含糊，那么在开发者邮递列表上询问。 因为可能说英语的最终用户也可能不明白它或者觉得它不好理解，所以最好改善这条消息。

50.2. 寄语程序员

50.2.1. 机理

本节描述如何在属于 PostgreSQL 版本的程序或者库里面支持本地语言。 目前它只适用于 C 语言。

向程序中增加 **NLS** 支持

1. 把下面的代码插入到程序的开头：

   ```
   #ifdef ENABLE_NLS
   #include &lt;locale.h&gt;
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("_progname_", LOCALEDIR);
   textdomain("_progname_");
   #endif
   ```

   （这里的 `_progname_` 实际上可以自由选择。）

2. 如果发现一条需要翻译的消息，那么就需要插入一个对 `gettext()` 的调用。比如

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   会改成

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   （如果没有配置 NLS ，那么 `gettext` 会定义成空操作。）

   这么干会出现一堆东西。一种常用的缩写是

   ```
   #define _(x) gettext(x)
   ```

   如果程序只通过一个或者少数几个函数做大部分的消息传递，比如后端里的 `ereport()` ，那么也可以用另外一个方法。在这些函数的内部对所有输入字符串调用 `gettext` 。

3. 在程序源代码所在的目录里加一个文件 `nls.mk` 。 这个文件将被当做 makefile 读取。在这里需要做下面一些变量的赋值：

   `CATALOG_NAME`

那些在 `textdomain()` 调用里提供的程序的名字。

`AVAIL_LANGUAGES`

提供的翻译的语言列表，开始的时候是空的。

`GETTEXT_FILES`

一列包含可翻译字符串的文件，也就是那些用 `gettext` 或者其它相应手段标记了的文件。 最终，这里会包括几乎所有的程序源文件。如果列表太长，你可以把第一个"文件"写成一个 `+` 和第二个词组成，第二个词是一个文件，在这个文件里每行包含一个文件名。

`GETTEXT_TRIGGERS`

生成给翻译者使用的消息表的工具，以便知道哪些函数调用包含可翻译字符串。 缺省时只知道 `gettext()` 调用。 如果你使用了 `_` 或其它标识符，那么你需要把它们列在这里。 如果可翻译字符串不是第一个参数，那么该项需要是这样的形式：`func:2` (用于第二给参数)。 如果函数支持多个消息，那么该项看起来就像这样: `func:1,2` (识别单个和多个消息参数)。

编译系统将自动制作和安装消息表。

50.2.2. 消息书写指导

这里是一些关于如何书写易于翻译的消息的指导：

- 不要在运行时构造语句，比如像

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  语句里这样的单词顺序会让其它语言很难翻译。 而且，即使你记得在每个片断上调用 `gettext()`，这些片断也不一定能很好地独立翻译。 最好复制一些代码，好让每条消息可以当作有机的整体进行翻译。 只有数字，文件名和类似的运行时变量才应该在运行时插入消息文本。

- 出于类似的原因，下面的东西不能用：

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  因为它假设了如何形成复数形。如果你看到这样的东西，你可以用下面方法解决

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

不过还是有让人失望的时候，有些语言在某些特殊规则上有超过两种形式。 通常最好是通过消息的设计避免这些东西，比如你可以这样写：

```
printf("number of copied files: %d", n);
```

如果你真的要构造一条恰当的复数形消息，也是支持的，但形式上有一点笨拙。 比如当通过 `ereport()` 生成一条概要或者详细消息时，你可以这样写：

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

第一个参数是对应英文单数形的格式字符串，第二个参数是对应英文复数形的格式字符串，第三个参数是决定是否是复数形的一个整数值。 随后的参数和通常一样对应与格式字符串中的参数值。（通常，控制复数形的值也是格式字符串中的其中一个参数。） 英语中只关心 `_n_` 是不是1，但其它语言中可能有多个复数形。 针对英文中作为一组的2个格式字符串，翻译者能够为根据 `_n_` 的运行值选中的恰当的那个提供多个替代字符串。

如果你需要一条不直接调用 `errmsg` 或 `errdetail` 的复数形消息， 必须使用下层的 `ngettext` 函数。具体参考gettext的文档。

- 如果你想和翻译者进行交流，比如说一条消息是如何与其它输出对齐的，那么在该字符串出现之前，放上一条以 `translator` 开头的注释，比如

```
/* translator: This message is not what it seems to be. */
```

这些注释都拷贝到消息表文件里，这样翻译者就可以看见它们了。

Chapter 51. 书写一个过程语言处理器

调用函数的时候，如果函数的书写语言不是目前的"版本-1"的编译语言接口 (这包括用户定义的过程语言写的函数，用SQL写的函数，以及用版本0的编译语言接口写的函数)，都会通过一个调用处理器处理具体的语言。 调用处理器有责任用一种有意义的方法执行这个函数，比如说解析所提供的文本等等。 本章简介如何书写一个新的过程语言调用处理器。

过程语言的调用处理器是一个"普通"的函数，必须使用一种编译语言来写，比如 C ，使用版本-1的接口，并且在PostgreSQL里注册成接受零个参数并且返回类型为 language_handler 。 这个特殊的伪类型标识该函数为一个调用处理器并且避免它直接在 SQL 命令中被调用。 关于C语言调用规范以及动态加载的更过细节请参考Section 35.9。

调用处理器的调用方式和其它函数一样：它接受一个指向一个 FunctionCallInfoData struct 的指针， 这个指针包含参数值和有关被调用的函数的信息，并且预期它返回一个 Datum 结果(如果它希望返回一个 SQL 的空结果，那么可能设置 isnull 字段)。 调用处理器和普通的被调函数的区别是 FunctionCallInfoData 结构的 flinfo->fn_oid 字段包含实际要调用的函数的 OID ， 而不是调用处理器自身。 调用处理器必须使用这个字段判断要执行哪个函数。通常， 传递进来的参数列表也是按照目标函数的声明设置的， 而不是给调用处理器设置的。

从系统表 pg_proc 里抓取函数入口以及分析被调函数的参数和返回类型就是调用处理器的事了。 来自 CREATE FUNCTION 命令中的 AS 子句将会在 pg_proc 行的 prosrc 字段中找到。 这个通常是过程语言本身的源文本，但也可以是别的东西，比如一个指向某个文件的路径名，或者任何告诉调用处理器如何详细处理的东西。

通常，每个 SQL 语句里面可能要调用同一个函数多次。 调用处理器可以利用 flinfo->fn_extra 字段避免重复地查找有关被调函数的信息。 这个字段初始为 NULL ，但是可以被调用处理器设置为指向有关被调函数的信息。 在随后的调用中，如果 flinfo->fn_extra 已经为非 NULL ，那么就可以直接使用它而免于重新查找信息。 调用处理器必须确保 flinfo->fn_extra 是用于指向一块至少会生存到当前查询结束的内存区里，因为一个 FmgrInfo 数据结构将会保存那么长的时间。 一个实现这个要求的方法是在 flinfo->fn_mcxt 声明的内存环境里分配一块额外的数据；这样的数据通常和 FmgrInfo 自己有一样的生命期。 但是处理器也可以同样选择使用一个更长生存期的环境，这样它就可以跨查询缓存函数定义。

在过程语言函数以触发器的形式调用的时候，就没有什么参数以通常的方式传递进来， 而是 FunctionCallInfoData 的 context 字段指向一个 TriggerData 结构，而不是像普通函数调用里面的 NULL 那样。 一个语言处理器应该为过程语言函数提供获取触发器信息的机制。

下面是一个用 C 写的存储过程语言处理器的模版：

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * 作为触发器过程调用
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * 作为函数调用
         */

        retval = ...
    }

    return retval;
}
```

在打点的地方放上几千行代码就可以完成调用处理器。

在把处理器函数编译成一个可加载的模块(参阅Section 35.9.6)之后，下面的命令就可以注册这个例子过程语言：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '_filename_'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

尽管提供一个调用处理器对创建一个最小的过程语言已经足够了，但是还有另外两个可选的函数如果也能提供的话将会让这个语言更加容易使用。 它们是有效性验证函数和内联处理器。 提供有效性验证函数将会允许CREATE FUNCTION时进行语言方面的检查。 提供内联处理器将会允许该语言支持通过DO命令执行的匿名代码块。

如果过程语言提供了有效性验证函数，必须将它声明为带单个 `oid` 类型参数的函数。 有效性验证函数的返回结果会被忽略，因此习惯上将其声明为返回 `void` 。 有效性验证函数在 `CREATE FUNCTION` 命令执行的末尾被调用，这时已经创建或者更新了一个使用这个过程语言

书写的函数。 传入的OID是这个函数在 `pg_proc` 中对应行的OID。 有效性验证函数必须以通常的方式获取这一行并做一些恰当的检查。 通常的检查包括：确认函数的参数和结果类型是被该语言支持的，函数体是符合该语言语法的。 如果有效性验证函数发现函数是正确的，只需要返回就可以了。 如果发现了错误应该通过 `ereport()` 报告错误。 抛出一个错误将强行回滚事务并避免了不正确的函数定义被提交。

有效性验证函数通常应该遵循check_function_bodies参数的设置。 如果被设置为无效，任何昂贵的或依赖于上下文的检查应该被跳过。 这个参数会被pg_dump关闭，这样就不用担心函数体对其他数据库中的对象有依赖关系了。 （因为这个要求，调用处理器不应该假定有效性验证函数已经做过了完整的检查。 有效性验证函数存在的目的并不是让调用处理器可以省略检查，而是为了在 `CREATE FUNCTION` 命令中包含明显的错误时可以立即通知用户） 精确的选择要检查哪些东西是由有效性验证函数决定的，注意当 `check_function_bodies` 有效时，核心的 `CREATE FUNCTION` 代码仅执行附加在函数上 `SET` 子句。 因此，当 `check_function_bodies` 无效时，很明显应该跳过哪些东西的结果将被GUC参数影响的检查，以避免加载dump时出现错误的失败。

如果过程语言提供了内联处理器，必须将它声明为带单个 `internal` 类型参数的函数。 内联处理器的返回结果会被忽略，因此习惯上将其声明为返回 `void` 。 当一个使用该语言的 `DO` 语句执行时，内联处理器将被调用。 实际传递的参数是一个 `InlineCodeBlock` 结构的指针。 这个结构包含了 `DO` 语句的参数信息，尤其是要执行的匿名代码块的文本。 内联处理器将执行这个代码并返回。

建议打包所有这些函数的声明以及 `CREATE LANGUAGE` 命令自身到一个扩展中， 这样只需要一个简单的 `CREATE EXTENSION` 命令就可以安装这个语言了。 关于书写扩展的方法，请参考Section 35.15 。

如果想书写自己的调用处理器，那么包含在标准发布里面的过程语言是很好的例子。参考一下源代码树中的 `src/pl` 子目录。 CREATE LANGUAGE参考页面中也有一些有用的信息。

Chapter 52. 写一个外数据包

Table of Contents

在外表上的所有操作都是通过它的外数据封装进行处理的， 它由核心服务器调用函数集组成。 外数据封装负责从远程数据源抓取数据，并且将它返回给PostgreSQL执行器。 如果支持更新外表，那么封装也必须处理。 本章概述了如何写新的外数据封装。

当尝试自己写的时候，在标准发布中的外数据包是好的参考。 查看下源代码树的 `contrib` 子目录。CREATE FOREIGN DATA WRAPPER 参考页也有一些有用细节。

> **Note:** SQL标准指定写外数据封装接口。然而，PostgreSQL不会实现API，因为努力调节它到PostgreSQL将是巨大的， 并且标准API没有获得广泛采用。

52.1. 外数据封装函数

FDW作者需要实现一个处理函数，并且任选一个验证函数。这两个函数必须写在一个编译语言里比如C， 使用版本-1接口。关于在C语言中调用约定和动态加载的详细细节， 参阅Section 35.9。

处理函数只返回调用函数的函数指针结构，其中通过规划器，执行器，以及各种维护命令 调用回调函数。写FDW的努力在于实现这些回调函数。 使用无参的PostgreSQL注册处理函数， 并且返回特殊伪类型 `fdw_handler` 。 回调函数是普通C函数，是不可见的或者在SQL级别上可随时调用的。 在Section 52.2中描述回调函数。

验证函数负责验证 由 `CREATE` 和 `ALTER` 命令为外数据封装，以及外服务器，用户映射和使用封装的外表 给定的选项，验证函数必须作为两个参数来注册，验证包含该选项的文本数组，并且该选项与表示对象类型的OID相关联（ 在系统目录OID形式下该对象被存储在， 要么 `ForeignDataWrapperRelationId` ， `ForeignServerRelationId` ， `UserMappingRelationId` ，或者 `ForeignTableRelationId` 中）。 如果不提供验证函数，则在对象创建时间或者对象变更时间不检查该选项。

52.2. 外数据封装回调程序

FDW处理函数返回包含指向下面描述的回调函数指针的palloc'd FdwRoutine 结构。 扫描相关函数是必须的，其余的是可选的。

在 src/include/foreign/fdwapi.h 中声明 FdwRoutine 结构类型， 参阅额外详情。

52.2.1. 扫描外表的FDW程序

```
void
GetForeignRelSize (PlannerInfo *root,
                   RelOptInfo *baserel,
                   Oid foreigntableid);
```

获得评估外表关系大小。这就是所谓的开始扫描外表的查询规划。 root 是关于查询的规划器的全局信息； baserel 是关于这个表的规划器信息； foreigntableid 是外表的 pg_class OID。 （ foreigntableid 可以从规划器数据结构中获得， 但是它明确被传递用来节省力气。）

在说明限制资格测试执行过滤之后， 该函数应该更新 baserel->rows 为通过表扫描返回的行期望数。 baserel->rows 的初始值仅仅是恒定缺省估计， 如果可能的话，这应该被替换。如果它可以对平均结果行宽度计算出更好的评估，那么 该函数可能也会选择更新 baserel->width 。

参阅Section 52.4可以获取额外信息。

```
void
GetForeignPaths (PlannerInfo *root,
                 RelOptInfo *baserel,
                 Oid foreigntableid);
```

创建外表扫描的可能访问路径。 这就是所谓的查询规划。 它被调用的参数和 GetForeignRelSize 相同。

该函数必须为外表扫描产生至少1个访问路径(ForeignPath 节点)而且 必须调用 add_path 添加每个这样的路径到 baserel->pathlist 中。 推荐使用 create_foreignscan_path 建立 ForeignPath 节点。 该函数可以生成多个访问路径，比如具有有效 pathkeys 表示预排序结果路径。 每个访问路径必须包含成本估计，并且包含需要标识具体预期扫描方法的任何FDW-私有信息。

参阅Section 52.4获取额外信息。

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid,
                ForeignPath *best_path,
                List *tlist,
                List *scan_clauses);
```

从所选择的外访问路径中创建 `ForeignScan` 规划节点。 这从查询规划结尾被调用。 该参数为 `GetForeignRelSize` ，加上所选择的 `ForeignPath` （通过 `GetForeignPaths` 预先生成）， 通过规划节点发出目标列表，并且限制子句通过规划节点被执行。

该函数必须创建并且返回 `ForeignScan` 规划节点； 推荐使用 `make_foreignscan` 建立 `ForeignScan` 节点。

参阅Section 52.4获取额外信息。

```
void
BeginForeignScan (ForeignScanState *node,
                  int eflags);
```

开始执行一个外部扫描。这是在执行器启动期间调用。它应该执行扫描开始前需要的任何初始化。 但没有开始执行实际扫描（应该执行第一次调用 `IterateForeignScan` ）。 `ForeignScanState` 节点已经被创建，但是它的 `fdw_state` 字段仍然是NULL。 通过 `ForeignScanState` 节点扫描的表信息（尤其是，来自底层的 `ForeignScan` 规划节点， 它包含任何通过 `GetForeignPlan` 提供的FDW-私有信息）。 `eflags` 包含描述该规划节点执行器的操作模式的标志位。

注意当 `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` 为真时， 该函数不应该执行任何外部可见行为； 它应该为 `ExplainForeignScan` 和 `EndForeignScan` 执行最小需求使得节点状态有效。

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

从外部源读取一行，在元组表槽中返回它（节点的 `ScanTupleSlot` 用于这个目的）。 如果没有更多行可用，那么返回NULL。元组表槽基础设施允许返回物理或者虚拟元组。 在大多数情况下后者选择从性能角度更可取。 注意这被称为在调用期间被重置的短暂内存语境。如果 你需要较长时间存储，或者使用节点的 `EState` 的 `es_query_cxt` ， 那么在 `BeginForeignScan` 中创建内存上下文。

返回的行必须匹配扫描外表的列标志。如果你选择优化掉不需要的列，那么 你应该在那些列位置插入空值。

注意PostgreSQL的执行器并不在乎返回的行是否违反任何在外表列定义的 `NOT NULL` 约束—但是规划器关心，如果 `NULL` 值出现在声明列中而不包含它们，那么可能错误地优化查询。 当用户声明不应该存在时，如果遇到 `NULL` 值，它可能会适当提高错误 （正如你需要在数据

类型不匹配的情况下执行）。

```
void
ReScanForeignScan (ForeignScanState *node);
```

从开始重启扫描。注意任何参数扫描取决于已改变的值， 因此扫描不一定返回完全相同的行。

```
void
EndForeignScan (ForeignScanState *node);
```

结束扫描并且释放资源。释放palloc内存往往不重要，但是比如打开文件并且链接远程服务器应该 被清理干净。

52.2.2. 更新外表FDW程序

如果FDW支持可写外表， 那么它应该提供一些或者所有下面的依赖于 FDW的需要和能力的回调函数：

```
void
AddForeignUpdateTargets (Query *parsetree,
                         RangeTblEntry *target_rte,
                         Relation target_relation);
```

在通过表扫描函数预先读取行之前执行 `UPDATE` 和 `DELETE` 操作。 FDW可能需要额外信息，比如行ID或者主键列值，为了确保它可以找到确切行更新或者删除。 为了支持它，该函数可以添加额外隐藏，或者"junk"，在 `UPDATE` 或者 `DELETE` 中从外表中检索列表中的目标列。

要做到这一点，添加 `TargetEntry` 项到 `parsetree->targetList` ，包含读取的额外值的表达式。 每个这样的项必须被标记 `resjunk` = `true` ， 并且有一个不同的 `resname` 在执行期间标识它。 避免使用匹配 `ctid``_N_` 或者 `wholerow``_N_` 的名称，正如核心系统可以 产生这些名字的垃圾列。

在改写过程中调用该函数，而不是规划器，因此该可用信息不同于可用的规划程序。 当 `target_rte` 和 `target_relation` 描述目标外表时， `parsetree` 是 `UPDATE` 或者 `DELETE` 命令的解析树。

如果 `AddForeignUpdateTargets` 指针被设置为 `NULL` ， 那么没有额外目标表达式被添加。 （这将不可能实现 `DELETE` 操作，尽管 `UPDATE` 可能仍然是可行的， 如果FDW依赖于一个标识行的未改变主键）。

```
List *
PlanForeignModify (PlannerInfo *root,
                   ModifyTable *plan,
                   Index resultRelation,
                   int subplan_index);
```

执行任何额外规划操作需要插入，更新或者删除外表。 该函数产生附属于执行更新操作的 ModifyTable 规划节点 的FDW-私有信息。这个私有信息必须有 List 形式，并且 在执行阶段将转交给 BeginForeignModify 。

root 是关于查询的规划器的全局信息。 plan 是 ModifyTable 规划节点， 除了 fdwPrivLists 字段外它是完整的。 resultRelation 通过射程表索引识别目标外表。 subplan_index 识别从零开始计算的 ModifyTable 规划节点是哪个目标； 如果你想要索引 plan->plans 或者其他 plan 节点的子结构，那么使用它。

参阅Section 52.4获取更多额外信息。

如果 PlanForeignModify 指针被设置为 NULL ， 没有采取额外规划时间操作，并且 fdw_private 列表转交给 BeginForeignModify 为零。

```
void
BeginForeignModify (ModifyTableState *mtstate,
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    int eflags);
```

开始执行一个外表修改操作。这个程序在执行器启动时调用。 应该在实际表修改前执行任何初始化。随后， ExecForeignInsert ， ExecForeignUpdate 或者 ExecForeignDelete 需要每个元组被插入，更新或者删除。

mtstate 是被执行的 ModifyTable 规划节点的整体状态； 关于规划的全局数据和执行状态通过该结构是可用的。 rinfo 是描述目标外表的 ResultRelInfo 结构。（ ResultRelInfo 的 ri_FdwState 字段用于FDW存储任何需要该操作的私有状态。） 如果任何的，那么 fdw_private 包含通过 PlanForeignModify 产生的私有数据。 subplan_index 识别 ModifyTable 规划节点是哪个目标。 eflags 包含描述这个规划节点的执行器操作模式的标志位。

注意当 (eflags & EXEC_FLAG_EXPLAIN_ONLY) 为真时， 该函数不应该执行任何外部可见操作； 它应该为 ExplainForeignModify 和 EndForeignModify 执行最小需求使得节点状态有效。

如果 BeginForeignModify 指针被设置为 NULL ， 那么在执行器启动期间不采取任何操作。

```
TupleTableSlot *
ExecForeignInsert (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

插入一个元组到外表。 `estate` 是查询的全局执行状态。 `rinfo` 是描述目标外表的 `ResultRelInfo` 结构。 `slot` 包含要插入的元组；它将匹配外表rowtype定义。 `planSlot` 包含通过 `ModifyTable` 规划节点的子计划产生的元组； 它不同于可能包含额外"junk"列的 `slot` 。

返回值要么是包含实际插入的数据的槽（这可能与提供的数据不同，比如作为触发器操作结果）， 如果没有行实际被插入，那么返回NULL（再次，通常作为触发器结果）。 传入的 `slot` 可以重新用于这个目的。

只有 `INSERT` 查询有 `RETURNING` 子句时， 才使用返回槽中的数据。因此，FDW可能选择优化返回依赖于 `RETURNING` 子句内容的一些或者全部列。 然而，必须返回一些插槽表示成功，或者查询报告的行数是错误的。

如果 `ExecForeignInsert` 指针被设置为 `NULL` ， 尝试插入外表将带有错误信息而失败。

```
TupleTableSlot *
ExecForeignUpdate (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

更新外表上的元组。 `estate` 是查询的全局执行状态。 `rinfo` 是描述目标外表的 `ResultRelInfo` 结构。 `slot` 包含元组的新数据； 它将匹配外表rowtype定义。 `planSlot` 包含通过 `ModifyTable` 规划节点的子计划产生的元组； 它不同于可能包含额外"junk"列的 `slot` 。 尤其是，通过 `AddForeignUpdateTargets` 请求的任何垃圾列将从该槽中提供。

返回值要么是包含实际更新的行的槽（这可能与提供的数据不同，比如作为触发器操作结果）， 如果没有行实际被更新，那么返回NULL（再次，通常作为触发器结果）。 传入的 `slot` 可以重新用于这个目的。

只有 `UPDATE` 查询有 `RETURNING` 子句时， 才使用返回槽中的数据。因此，FDW可能选择优化返回依赖于 `RETURNING` 子句内容的一些或者全部列。 然而，必须返回一些插槽表示成功，或者查询报告的行数是错误的。

如果 `ExecForeignUpdate` 指针被设置为 `NULL` ， 尝试更新外表将带有错误信息而失败。

```
TupleTableSlot *
ExecForeignDelete (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

删除外表上的元组。 `estate` 是查询的全局执行状态。 `rinfo` 是描述目标外表的 `ResultRelInfo` 结构。 `slot` 不包含任何有用调用， 但是可以用于保留返回的元组。 `planSlot` 包含通过 `ModifyTable` 规划节点的子计划产生的元组；尤其是，它将具有通过 `AddForeignUpdateTargets` 请求的任何垃圾列。 垃圾列必须用于标识要删除的元组。

返回值要么是包含实际被删除行的槽， 如果没有行实际被删除，那么返回NULL（再次，通常作为触发器结果）。 传入的 `slot` 可以用于保留返回的元组。

只有 `DELETE` 查询有 `RETURNING` 子句时， 才使用返回槽中的数据。因此，FDW可能选择优化返回依赖于 `RETURNING` 子句内容的一些或者全部列。 然而，必须返回一些插槽表示成功，或者查询报告的行数是错误的。

如果 `ExecForeignDelete` 指针被设置为 `NULL`， 尝试删除外表将带有错误信息而失败。

```
void
EndForeignModify (EState *estate,
                  ResultRelInfo *rinfo);
```

结束表更新并且释放资源。释放palloc内存往往不重要，但是比如打开文件并且链接远程服务器应该 被清理干净。

如果 `EndForeignModify` 指针被设置为 `NULL`， 那么在执行器关闭期间不采取任何操作。

```
int
IsForeignRelUpdatable (Relation rel);
```

报告指定外表支持的更新操作。返回值应该是规则事件数的位掩码，标志着 使用 `CmdType` 枚举通过外表支持的操作；即 `(1 << CMD_UPDATE) = 4` 为 `UPDATE`， `(1 << CMD_INSERT) = 8` 为 `INSERT` 并且 `(1 << CMD_DELETE) = 16` 为 `DELETE`。

如果 `IsForeignRelUpdatable` 指针被设置为 `NULL`，那么FDW分别 提供 `ExecForeignInsert`， `ExecForeignUpdate` 或者 `ExecForeignDelete`，那么 外表被认为是可插入，可更新或者可删除的。 如果FDW支持一些可更新的和一些不可更新的表，那么需要这个函数。 （即使这样，它允许在执行程序中抛出错误而不是在这个函数中进行检查。然而，该函数 用于在 `information_schema` 视图中显示可更新。）

52.2.3. `EXPLAIN` 的FDW程序

```
void
ExplainForeignScan (ForeignScanState *node,
                    ExplainState *es);
```

为外表扫描打印额外 `EXPLAIN` 输出。 该函数可以调用 `ExplainPropertyText` 和相关函数添加到 `EXPLAIN` 输出字段。 `es` 中的标志位可以用于决定打印什么，并且检查 `ForeignScanState` 节点状态 用来在 `EXPLAIN ANALYZE` 情况下提供运行时统计。

如果 `ExplainForeignScan` 指针被设置为 `NULL`，那么 在 `EXPLAIN` 期间不打印额外信息。

```
void
ExplainForeignModify (ModifyTableState *mtstate,
                      ResultRelInfo *rinfo,
                      List *fdw_private,
                      int subplan_index,
                      struct ExplainState *es);
```

为外表更新打印额外 `EXPLAIN` 输出。 该函数可以调用 `ExplainPropertyText` 和相关函数添加到 `EXPLAIN` 输出字段。 `es` 中的标志位可以用于决定打印什么，并且检查 `ModifyTableState` 节点状态 用来在 `EXPLAIN ANALYZE` 情况下提供运行时统计。前四个参数为 `BeginForeignModify` 是相同的。

如果 `ExplainForeignModify` 指针被设置为 `NULL` ， 那么 在 `EXPLAIN` 期间不打印任何额外信息。

52.2.4. `ANALYZE` 的FDW程序

```
bool
AnalyzeForeignTable (Relation relation,
                     AcquireSampleRowsFunc *func,
                     BlockNumber *totalpages);
```

当在外表上执行ANALYZE时，调用该函数。 如果FDW可以收集外表的统计，它应该返回 `真` ， 并且提供一个指针给函数，该函数从 `func` 中的表中收集样本行。 以及 `totalpages` 的页中表的估计大小。 否则，返回 `false` 。

如果FDW不支持任何表的统计，那么 `AnalyzeForeignTable` 指针可以设置为 `NULL` 。

如果提供，那么样本收集函数必须有识别标志

```
int
AcquireSampleRowsFunc (Relation relation, int elevel,
                       HeapTuple *rows, int targrows,
                       double *totalrows,
                       double *totaldeadrows);
```

应该从表中收集达到 `targrows` 行的随机抽样调查，并且存储到 调用者提供的 `rows` 数组。必须返回收集行的真实数。 此外，将表中死的和活行 总数估计存储到输出参数 `totalrows` 和 `totaldeadrows` 中。 （如果FDW 没有死行的任何概念，那么设置 `totaldeadrows` 为零。）

52.3. 外数据封装辅助函数

从核心服务器输出一些辅助函数，因此外数据封装作者可以很容易访问FDW-相关对象的属性，比如FDW选项。 为了使用任何这些函数，你需要包含你的源文件中的头文件 `foreign/foreign.h` 。 该头文件也定义了这些函数返回的结构体类型。

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

该函数为给定OID的外数据封装返回一个`ForeignDataWrapper`对象。 `ForeignDataWrapper`对象包含 FDW的属性 (参阅`foreign/foreign.h`获取更多详情）。

```
ForeignServer *
GetForeignServer(Oid serverid);
```

该函数为给定OID的外服务器返回一个`ForeignServer`对象。 `ForeignServer`对象包含 服务器的属性 (参阅`foreign/foreign.h`获取更多详情）。

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

该函数为给定服务器上给定角色的用户映射返回`UserMapping`对象。 （如果没有特定用户映射，那么它将返回`PUBLIC`映射，或者如果什么没有，则抛出错误。） `UserMapping`对象包含用户映射属性(参阅`foreign/foreign.h` 获取详情）。

```
ForeignTable *
GetForeignTable(Oid relid);
```

该函数为给定OID外表返回`ForeignTable`对象。 `ForeignTable`对象包含外表属性(参阅`foreign/foreign.h`获取更多详情)。

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

该函数返回在列表`DefElem`形式中具有给定外表OID和属性数每列FDW选项， 如果该列没有选项，那么返回零。

一些对象类型除了基于OID的那个有基于名字的查找函数：

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

该函数返回具有给定名称的外数据封装的`ForeignDataWrapper`对象。 如果没有找到封装，如果missing_ok是真的，那么返回NULL，否则抛出错误。

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

该函数为给定名字的外服务器返回`ForeignServer`对象。 如果没有找到服务器，如果missing_ok是真的，那么返回NULL，否则抛出错误。

52.4. 外数据封装查询规划

FDW回调函数 `GetForeignRelSize` , `GetForeignPaths` , `GetForeignPlan` 和 `PlanForeignModify` 必须适合PostgreSQL规划器的工作。 这有一些他们必须做的说明。

在 `root` 和 `baserel` 中的信息可以用于减少从外表中（因此降低成本）读取的信息量。 baserel->baserestrictinfo 特别有趣，正如它包含应该用于过滤读取的行的限制资格测试 (`WHERE` clauses)。 （FDW本身不需要执行这些测试，正如核心执行者反而可以检查它们。） baserel->reltargetlist 可以用于决定需要抓取哪个列；但是注意它仅仅列出通过 `ForeignScan` 规划节点发出的列，不是在质量评估中使用的列也不是查询输出列。

各种私有字段可用于FDW规划函数保持信息。一般来说，无论在FDW私有字段存储什么 应该是palloc，所以它将在规划结束后被回收。

baserel->fdw_private 是 空 指针，可用于FDW规划函数存储与特定外表相关的 信息。当创建 `baserel` 结点时，核心规划器并不接触它除了初始化为NULL外。 从 `GetForeignRelSize` 到 `GetForeignPaths` 和/或者 `GetForeignPaths` 到 `GetForeignPlan` 传递信息是非常有用的， 从而避免重复计算。

`GetForeignPaths` 可以通过在 `ForeignPath` 节点的 `fdw_private` 字段存储私有信息标识不同访问路径的含义。 作为 `List` 指针声明 `fdw_private` ，但是实际上可以包含任何由于 核心规划器没有接触的东西。然而，最好办法是通过 `nodeToString` 可倾式表示形式， 用于调试后端可用支持。

`GetForeignPlan` 可以检查所选的 `ForeignPath` 节点的 `fdw_private` 字段，并且可以产生放在 `ForeignScan` 规划节点中的 `fdw_exprs` 和 `fdw_private` 列表， 在执行时间他们可用。这些列表必须在 `copyObject` 知道如何拷贝的形式中被表示。 `fdw_private` 列表没有其他限制，并且不能通过任何方式的核心后端进行解释。 `fdw_exprs` 列表如果不是零，那么希望包含在运行时执行的表达式树。 这些树将通过规划器使得它们完全可执行进行后处理。

在 `GetForeignPlan` 中，往往目标列表可以拷贝到规划节点中。 已通过的scan_clauses列表包含和 baserel->baserestrictinfo 相同分句， 但是为了更好的执行效率可能重新排序。 在简单情况下FDW可以从scan_clauses列表（使用 `extract_actual_clauses` ）中删除 `RestrictInfo` 节点， 并且将所有分句放入规划节点的资格列表中，这意味着在运行期间通过执行器将检查所有分句。 更多复杂的FDW可能检查一些内部分句，在这种情况下那些分句可以从规划节点的列表中删除，以便执行器 不会浪费复查它们的时间。

作为一个例子，FDW可能标识一些来自 `_foreign_variable_` `=` `_sub_expression_` 的限制分句，它决定了在给定 `_sub_expression_` 的本地评估值的 远程服务器上被执行。 在 `GetForeignPaths` 期间产生了该分句的实际标识， 因为它会影响路径的成本估算。路径的 `fdw_private` 字段可能包含一个指向 已标识分句的 `RestrictInfo` 节点的指针。然后 `GetForeignPlan` 从scan*clauses*中 删除该分句，但是增加

`_sub_expression` 到 fdw*exprs* 以确保 它获得可执行形式。它也可能将控制信息放到规划节点的 *fdw_private* 字段以告知执行函数在运行时间执行什么。传送到远程服务器的查询可能涉及到像 *WHERE* *_foreign_variable* fdw_exprs`表达式树评估中获得的参数值。

FDW应该总是构建至少一个仅仅依赖于表的限制分句的路径，在连接查询中，它也可能选择构建依赖于连接分句的路径， 比如 `_foreign_variable_` `=` `_local_variable_` 。该分句没有在 `baserel->baserestrictinfo` 中找到， 但是必须在关系的连接列表中寻找。 使用分句的路径被称为"参数化路径"。 它必须标识用于使用 `param_info` 合适值的已选择连接分句的其他关系； 使用 `get_baserel_parampathinfo` 计算该值。在 `GetForeignPlan` 中， 连接分句的 `_local_variable_` 部分将被添加到 `fdw_exprs` 中， 然后运行时该情况与普通限制分句一样运行。

当规划一个 `UPDATE` 或者 `DELETE` 的时候， `PlanForeignModify` 可以查找外表的 `RelOptInfo` 结构， 并且充分使用通过扫描规划函数预先创建的 `baserel->fdw_private` 数据。 然而，在 `INSERT` 中不扫描目标表，所以没有 `RelOptInfo` 。 通过 `PlanForeignModify` 返回的 `List` 与 `ForeignScan` 规划节点 的 `fdw_private` 列表有相同的限制， 即它必须包含 `copyObject` 知道如何拷贝的结构。

`UPDATE` 或者 `DELETE` 反对外部数据源支持并发更新，推荐 `ForeignScan` 操作锁定抓取的行，也许通过 `SELECT FOR UPDATE` 的等价物。 当在 `SELECT FOR UPDATE/SHARE` 引用外表时，FDW可能也会在抓取时选择锁定行。 如果没有做，就有关外表来说， 那么 `FOR UPDATE` 或者 `FOR SHARE` 选项本质上是无操作的。 这种操作可能会产生本地表操作上轻微的语义差异，行锁定通常尽可能的延迟： 远程行可能获得锁定即使他们随后没有本地应用限制或者连接条件。 然而，匹配局部语义确实需要每行的额外远程访问，并且不可能依赖于 外部数据源提供的锁定语义内容。

Chapter 53. 基因查询优化器

Table of Contents

> 作者: 由Martin Utesch（`<[utesch@aut.tu-freiberg.de](mailto:utesch@aut.tu-freiberg.de)>`）为德国弗来堡矿业及技术大学自动控制系写作。

53.1. 作为复杂优化问题的查询处理

在所有关系型操作符里，最难以处理和优化的一个是连接。 一个查询需要回答的可选规划的数目将随着该查询包含的连接的个数呈指数增长。 在访问关系分支时的进一步优化措施是由多种多样的连接方法 (例如嵌套循环、索引扫描、融合连接等)来支持处理独立的连接和多种多样的索引 (比如PostgreSQL中的 B-tree, hash, GiST和 GIN)。

目前PostgreSQL优化器的实现在候选策略空间里执行一个近似穷举搜索。 这个算法最早是在IBM System R database 数据库中引入的，它生成一个近乎最优的连接顺序， 但是如果查询中的连接增长得很大，它可能会消耗大量的时间和内存空间。 这样就使普通的PostgreSQL查询优化器不适合那种连接了大量表的查询。

德国弗来堡矿业及技术大学自动控制系的成员在试图把PostgreSQL 作为用于一个电力网维护中做决策支持的知识库系统的后端时，碰到了上面的问题。 该 DBMS 需要为知识库系统的推导机处理很大的连接查询。 在可能的查询规划空间里进行检索的恶劣性能引起了人们对发展新的优化技术的需求。

在随后的内容里，描述了一个基因算法，这个算法用一种对涉及大量连接的查询很有效的方式解决了连接顺序的问题。

53.2. 基因算法

基因算法(GA)是一种启发式的优化法，它是通过不确定的随机搜索进行操作。 优化问题的可能解的集合被认为是个体组成的种群。 一个个体对它的环境的适应程度由它的适应性表示。

一个个体在搜索空间里的参照物用染色体表示(实际上是一套字符串)。 一个基因是染色体的一个片段，基因是被优化的单个参数的编码。 对一个基因的典型的编码可以是二进制或整数。

通过仿真进化过程的重组、变异、选择找到新一代的搜索点，它们的平均适应性要比它们的祖先好。

根据comp.ai.genetic FAQ，不论怎么强调GA在解决一个问题时不是纯随机搜索都不过份。 GA使用随机处理，但是结果明显不是随机的(比随机更好)。

Figure 53-1. 基因算法的结构化框图

P(t)	时刻 **t** 的父代
P''(t)	时刻 t 的子代

```
+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evaluate FITNESS of P(t)                |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evaluate FITNESS of P''(t)          |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

53.3. PostgreSQL 里的基因查询优化(GEQO)

GEQO模块是试图解决类似著名的旅行商问题(TSP)的查询优化问题的。 可能的查询规划被当作整数字符串进行编码。每个字符串代表查询里面一个关系到下一个关系的连接的顺序。例如，下面的连接树

```
     /\
    /\ 2
   /\ 3
  4  1
```

是用整数字符串'4-1-3-2'编码的，这就是说，首先连接关系'4'和'1'，然后'3'，然后是'2'， 这里的 1, 2, 3, 4 都是PostgreSQL优化器里的关系标识(ID)。

在PostgreSQL里的GEQO实现的一些特性是：

- 使用稳定状态的 GA(替换全体中最小适应性的个体，而不是整代的替换) 允许向改进了的查询规划快速逼近。这一点对在合理时间内处理查询是非常重要的；
- 边缘重组交叉的使用特别适于在用GA解决TSP问题时保持边缘损失最低。
- 否决了把突变作为基因操作符的做法，这样生成合法的TSP漫游时不需要修复机制。

GEQO模块的一部分是采用的 D.Whitley 的 Genitor 算法。

GEQO模块让PostgreSQL查询优化器可以通过非穷举搜索有效地支持大的连接查询。

53.3.1. GEQO生成的候选规划

GEQO规划过程使用标准的规划器代码生成各个独立表的扫描规划。 然后用遗传算法生成连接规划。正如上面讲到的，每个候选的连接规划由基本表的连接序列表示。 在初始阶段，GEQO简单的生成一些随机的连接序列。 对每个连接序列，调用标准的规划器代码评估以这样的连接序列执行查询的成本。 （对连接序列的每一个步骤，考虑所有可能的三种连接策略；并且所有初期决定的关系扫描规划也是有效的。 评估值是所有这些可能性中最廉价的。） 评估成本更低的连接序列被认为比代价高的那些连接序列"更加合适"。基因算法淘汰那些最不合适的候选。 然后通过组合最合适的基因生成新的候选，即通过随机选择一部分低成本的连接序列去产生作为候选的新的序列。 这个过程不断重复直到考虑过的连接序列达到一个预设的值，然后在搜索期间找到的最好的结果作为最终生成的执行计划。

这个过程天生是不确定的，因为在选择初期总群以及随后对最佳候补实施"突变"时都带有随机性。 为了避免被选中的计划发生令人惊讶的变化，每次GEQO算法执行都根据当前geqo_seed参数的设置开始一个随机数生成器。 只要 `geqo_seed` 以及其他GEQO参数保持

固定，对一个给定的查询(以及其他规划器输入，比如统计信息)将生成相同的执行计划。 如果要尝试不同的搜索路径，可以尝试改变 `geqo_seed` 。

53.3.2. PostgreSQL GEQO未来的实现任务

还需要一些工作来改进基因算法的参数设置。 在文件 `src/backend/optimizer/geqo/geqo_main.c` 里的过程 `gimme_pool_size` 和 `gimme_number_generations` 在设置参数时不得不为两个竞争需求做出折衷：

- 查询规划的优化
- 计算处理时间

在当前的实现中，每个候选连接序列的适应性是通过运行标准规划器的连接选择和代价评估代码从头开始评估的。 当不同的候选使用相似的连接子序列，将有大量的工作会重复。 通过保持子序列的代价评估值可以显著的提升速度。问题在于要避免为了保持这个状态而不合理的内存扩展。

在最基本的层面上，并不清楚用给 TSP 涉及的 GA 算法解决查询优化的问题是否合适。 在 TSP 的情况下，与任何子字符串(部分旅游)相关的开销都是独立于旅游的其它部分的，但是目前，这一点对于查询优化是不同的。 因此，可以怀疑边缘重组交叉是否最有效的突变过程。

53.4. 进一步阅读

下面的资源包含一些额外的关于基因算法的信息：

- The Hitch-Hiker's Guide to Evolutionary Computation, (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design, by Craig Reynolds
- 数据库系统原理
- POSTGRES查询优化器的设计和实现

Chapter 54. 索引访问方法接口定义

Table of Contents

本章定义PostgreSQL核心系统和索引访问方法 之间的接口，后者管理独立的索引类型。 除了在这里声明的东西之外，核心系统对索引一无所知，因此可以通过书写累加上来的代码，开发一种完全新的索引类型。

PostgreSQL里的所有索引技术上都叫做 从属索引，也就是说，索引在物理上是与它描述的表文件分离的。 每个索引是以其自己的物理关系的方式存储的，因此它们也在 `pg_class` 系统表里面有记录描述。 一个索引的内容是完全在其索引访问方法的控制之下的。 实际上，所有索引访问方法都把索引分裂成标准大小的页面，这样他们就可以使用普通的存储管理器和缓冲区管理器来访问索引的内容了。 (所有现有的索引访问方法更是使用Section 58.6 里面描述的标准的页面布局，并且索引行头都使用同样的格式；但是这些东西都不是强制访问方法执行的。 也就是说必要的话你可以不用这些标准格式。)

索引实际上是一些数据的键值与索引的父表中的行版本(元组)的行标识符或TID之间的映射。 一个 TID 由一个块号和一个该块内的项编号组成(参阅Section 58.6)。 这些就是从该表中抓取某个特定行版本所需的足够的信息。 索引并不直接知道在 MVCC 下，同一个逻辑行可能有多个现存的版本；对于索引而言，每个元组都是一个独立的对象，都需要自己的索引项。 因此，对一行的更新总是为该行创建全新的索引项，即使键值没有改变也如此（HOT元组是个例外，但索引并不处理这些）。 已经废弃的元组的索引项是在废弃元组自己被回收的时候回收(通过vacuum)。

54.1. 索引的系统表记录

每个索引访问方法都在系统表 `pg_am` 里面用一行来描述(参阅Section 47.3)。 一个 `pg_am` 行的主要内容是引用 `pg_proc` 里面的记录，用来标识索引访问方法提供的索引访问函数。 这些函数的接口(API)在本章后面描述。 另外， `pg_am` 的数据行声明了几个索引访问方法的固定属性，比如，它是否支持多字段索引。 目前还没有创建、删除 `pg_am` 记录的特殊支持；任何想写这么一个新的访问方法的人都需要能够自己向这个表里面插入合适的新行。

要想有真正用处，一个索引访问方法还必须有一个或多个操作符族和操作符类，定义在 `pg_opfamily`，`pg_opclass`，`pg_amop` 和 `pg_amproc` 里面。 这些记录允许规划器判断哪些查询的条件可以适用于用这个索引访问方法创建的索引。 操作符族和操作符类在Section 35.14里面定义，是读取本章的前提之一。

一个独立的索引是由一行 `pg_class` 记录以物理关系的方式描述的，加上一个 `pg_index` 行，表示该索引的逻辑内容 —也就是说，它所拥有的索引字段集，以及被相关的操作符类捕获的这些字段的语义。 索引字段(键值)可以是下层表的字段，也可以是该表的数据行上的表达式。 索引访问方法通常不关心索引的键值来自哪里(它总是操作经过预计算的键值)，但是它会对 `pg_index` 里面的操作符类信息很感兴趣。 所有这些系统表记录都可以当作 `Relation` 数据结构的一部分访问，这个数据结构会被传递到对该索引的所有操作上。

`pg_am` 中的有些标志字段的含义并不那么直观。 `amcanunique` 的需求在Section 54.5里讨论， `amcanmulticol` 标志断言该索引访问方法支持多字段索引， `amoptionalkey` 断言它允许对那种在第一个索引字段上没有给出可索引限制子句的扫描。 如果 `amcanmulticol` 为假，那么 `amoptionalkey` 实际上说的是该访问方法是否允许不带限制子句的全索引扫描。 那些支持多字段索引的访问访法必须支持那些在省略了除第一个字段以外的其它字段的约束的扫描；不过，系统允许这些访问访法要求在第一个字段上出现一些限制，这一点是通过把 amoptionalkey 设置为假来实现的。 一个访问访法可能设置 `amoptionalkey` 为假,如果它不索引NULL值的话。 因为大多数可以索引的操作符都是严格的，因此不能对 NULL 输入返回 TRUE， 所以，第一眼看见会觉得不为 NULL 存储索引记录的想法很吸引人：因为他们不可能被一个索引扫描返回。 不过，这个想法在一个给出的索引字段上没有限制子句的索引扫描的情况下就不行了；这样的扫描应该包括 NULL 行。 实际上，这意味着设置了 `amoptionalkey` 为真的索引必须索引 NULL值， 因为规划器可能会决定在根本没有扫描键字的时候使用这样的索引。 这样的索引必须可以在完全没有扫描键字的情况下运行。 另外一个限制是一个支持多字段索引的索引访问方法必须索引第一个字段后面的字段的 NULL值，因为规划器会认为这个索引可以用于那些没有限制这些字段的查询。 比如，假设有个在(a,b)上的索引，而一个查询的条件是 `WHERE a = 4` 。系统会认为这个索引可以用于扫描 `a = 4` 的数据行，如果索引忽略了 `b` 为空的数据行，那么就是错误的。 不过，如果第一个索引字段值是空，那么忽略它是 OK 的。 一个索引NULL值的索引访问方法可能会设置 `amsearchnulls` ， 表明它支持 `IS NULL` 和 `IS NOT NULL` 子句作为搜索条件。

54.2. 索引访问方法函数

索引访问方法必须提供的索引构造和维护函数有：

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

创建一个新索引。索引关系已经物理上创建好了，但是是空的。 必须用索引访问方法要求的固定数据和代表所有已经在表里的行的数据项填充它。 通常， `ambuild` 函数会调用 `IndexBuildHeapScan()` 扫描该表以获取现有行并计算需要插入索引的键字。

```
void
ambuildempty (Relation indexRelation);
```

创建一个空的索引，并写到给定关系的初始fork(INIT_FORKNUM)中。 这个方法只会为unlogged表调用;在服务器重启动时写入到初始fork的空索引会被复制到主关系。

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique);
```

向现有索引插入一个新行。 `values` 和 `isnull` 数组给出需要制作索引的键字值，而 `heap_tid` 是要被索引的 TID 。 如果该访问方法支持唯一索引(它的 `pg_am` . `amcanunique` 标志是真)， 那么 `checkUnique` 指示需要执行这种唯一性检查。 具体情况依赖于该唯一制约是否是可延期的（deferrable）而不同; 详细请参阅 Section 54.5。 通常访问方法只有在执行唯一性检查时才需要 `heapRelation` 参数(它将深入到heap中确认行是否是活的)。

只有当 `checkUnique` 是 `UNIQUE_CHECK_PARTIAL` 时，该函数的返回值才有意义。 TRUE的返回值意味着新的索引项是唯一的，FALSE意味着不唯一（并且延期的唯一性检查必须被调度）。 其它情况下推荐返回常量FALSE。

一些索引可能并不索引所有的行。如果行不被索引， `aminsert` 应该不做任何事情就直接返回。

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

从索引中删除行。 这是一个"批量删除"的操作，通常都是通过扫描整个索引，检查每条记录，看看它是否需要被删除来实现的。 可以调用传递进来的 `callback` 函数，调用风格是： ``` callback(``_TID_ ``` , callback_state) returns bool， 其作用是判断某个用其引用的 TID 标识的索引项是否需要删除。 必须返回 NULL 或者是一个 palloc 出来的，包含删除操作执行影响的统计信息的结构。 如果不需要向 `amvacuumcleanup` 传递信息，返回 NULL 也是 OK 的。

由于 `maintenance_work_mem` 的限制，在删除多行的时候 `ambulkdelete` 可能需要被调用多次， `stats` 参数是先前在这个索引上的调用结果(在一个 `VACUUM` 操作内部第一次调用的话则是 NULL)。 这将允许 AM 在整个操作过程中积累统计信息。 典型的，如果传递的 `stats` 不是 null 的话， `ambulkdelete` 将会修改并返回相同的结构。

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

在一个 `VACUUM` 操作(一个或多个 `ambulkdelete` 调用)之后清理。 虽然不必做任何返回索引状态之外的任何其他事情，但是它通常用于批量清理，比如说回收空的索引页面。 `stats` 是最后的 `ambulkdelete` 调用返回的东西或者 NULL(如果因为没有行需要删除而未调用 `ambulkdelete` 的话)。 如果结果不是 NULL，那么它必须是一个 palloc 出来的结构。 它包含的统计信息将用于更新 `pg_class` 并且由 `VACUUM` 报告(如果给出了 `VERBOSE`)。 如果索引在 `VACUUM` 操作的过程中根本没有改变，那么返回 NULL 也是 OK 的，否则必须返回当前状态。

在PostgreSQL8.4中， `amvacuumcleanup` 也会在 `ANALYZE` 完成时被调用。 这时， `stats` 总是为NULL，并且返回值会被忽略。 通过检查 `info->analyze_only` 可以区分出这种情况。 建议访问方法在这样的调用里除了插入后的清理不要做其他事情，并且这只在autovacuum工作进程中。

```
bool
amcanreturn (Relation indexRelation);
```

检查索引是否支持*index-only*扫描，通过为一个索引项以IndexTuple的形式返回被索引的列值。 如果支持返回TRUE，否则返回FALSE。 如果索引AM永远不支持index-only扫描（比如hash，它只存储哈希值而不是原始数据）， 可以有充分的理由把 `pg_am` 中的 `amcanreturn` 字段设置为零。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

估算一个索引扫描的开销。该函数在下面的Section 54.6中有详细的讨论。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

为一个索引分析和验证 reloptions 数组，仅当一个索引存在非空 reloptions 数组时才会被调用。 `reloptions` 是一个 `text` 数组，包含 `_name_``=``_value_` 格式的项。 该函数应当创建一个 `bytea` 值，该值将被拷贝进索引的 relcache 项的 `rd_options` 字段。 `bytea` 值的数据内容可以由访问方法定义，不过目前所有的标准访问方法都使用 `StdRdOptions` 结构。当 `validate` 为真时，如果任何一个选项不可识别或者含有非法值，该函数都应当报告一个适当的错误信息；当 `validate` 为假时，非法项应该被悄悄的忽略。 （当载入已经存储在 `pg_catalog` 中的选项时， `validate` 为假，仅在访问方法已经改变了选项规则的时候才可能找到非法项，在此情况下可以忽略废弃的项。） 如果默认行为正是想要的，那么返回 NULL 也 OK 。

索引的目的当然是支持那些包含一个可以索引的 `WHERE` 条件的行的扫描，这个条件通常叫修饰词或扫描键字。 索引扫描的语义在下面的Section 54.3里面有更完整的描述。 一个索引访问方法可以支持"plain"索引扫描，"bitmap"索引扫描，或者两者都支持。 必须或可以提供的与扫描有关的函数有：

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

准备一个索引扫描。 `nkeys` 和 `norderbys` 参素指示扫描中使用的修饰词和排序操作符的个数；它们可能对空间分配有用。 注意实际的扫描键还没有提供。 结果必须是一个 palloc 出来的结构。 由于实现的原因，索引访问方法必须通过调用 `RelationGetIndexScan()` 来创建这个结构。 在大多数情况下， `ambeginscan` 本身除了调用上面这个函数和可能获取一些锁之外几乎不干别的事情；索引扫描启动时的有趣部分在 `amrescan` 里。

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

启动或重新启动一个索引扫描，可能会使用新的扫描键字。 （要使用先前提供的键重启动，给 `keys` 和/或者 `orderbys` 传入NULL） 记住扫描键字或排序操作符的个数不予许大于传给 `ambeginscan` 的数值。 实际上，重新启动特性用于这样的场景：当一个新的外元组被嵌套循环(nested-loop)连接选中时，需要一个新的键比较值，但是扫描键结构仍然是相同的。

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

在给出的扫描里抓取下一个行，向给出的方向移动(在索引里向前或者向后)。 如果抓取到了行，则返回 TRUE ，如果没有抓到匹配的行，返回 FALSE 。 在为 TRUE 的时候，该行的 TID 存储在 `scan` 结构里。 请注意"成功"只是意味着索引包含一个匹配扫描键字的条目，并不是说该行仍然在堆中存在，或者是能够通过调用者的快照检查(译注：MVCC 快照，用于判断事务边界内的行可视性)。 如果成功， `amgettuple` 必须设置 `scan->xs_recheck` 为TRUE或FALSE。 FALSE意味着已经可以确定索引项匹配扫描键字。 TRUE意味着尚不确定，在取到堆元组后必须对堆元组再次检查代表这个扫描键值的条件。

如果索引支持index-only扫描（比如， `amcanreturn` 为它返回TRUE）， 那么成功执行后，这个AM也必须检查 `scan->xs_want_itup` ， 如果为TRUE, 它必须通过存储在 `scan->xs_itup` 中的 `IndexTuple` 指针以及元组描述符 `scan->xs_itupdesc` 为这个索引项返回原始的被索引数据。 （访问方法需要负责维护被这个指针引用的数据。至少在该扫描下一次调用 `amgettuple` ， `amrescan` 或 `amendscan` 前，这个数据必须保持完好）

如果访问方法支持"plain"索引扫描，只需要提供 `amgettuple` 函数。 如果不是，在它的 `pg_am` 行中的 `amgettuple` 字段必须被设置成零。

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

在指定的扫描中抓取所有元组并把它们加入到调用者提供的 `TIDBitmap` 中（换句话说，元组的ID集合加入到某个已存在的bitmap）。 函数返回抓取到的元组数（这可能只是一个近似计数，某些AM实例并不检测重复）。 当插入元组的TID到bitmap, `amgetbitmap` 可以指示对特定的元组TID需要对扫描条件做再检查。 这和 `amgettuple` 函数的输出参数 `xs_recheck` 类似。 注意：在当前实现中，支持这个特性涉及到支持bitmap自身的有损存储，因此调用者为可再检查的元组再次检查扫描条件和部分索引谓词（如果有的话）。 然而，这可能不会总是正确的。 `amgetbitmap` 和 `amgettuple` 不能在同一个索引扫描中使用；在使用 `amgetbitmap` 的时候还有其它限制，在Section 54.3里给出解释。

只有访问方法支持"bitmap"索引扫描时才需要提供 `amgetbitmap` 函数。 如果访问方法不支持的话，必须在它的 `pg_am` 行里设置 `amgetbitmap` 字段为零。

```
void
amendscan (IndexScanDesc scan);
```

结束扫描并释放资源。不应该释放 `scan` 本身，但访问方法内部使用的任何锁或者销(pin)都应该释放。

```
void
ammarkpos (IndexScanDesc scan);
```

标记当前扫描位置。访问方法只需要支持每次扫描里面有一个被记住的扫描位置。

```
void
amrestrpos (IndexScanDesc scan);
```

把扫描恢复到最近标记的位置。

通常，任何索引访问方法函数的 `pg_proc` 记录都应该显示正确数目的参数， 只是把类型都声明为类型 `internal` (因为大多数参数的类型都是 SQL 不识别的类型，并且不希望用户直接调用该函数)。 返回类型根据具体情况声明为 `void` , `internal` , or `boolean` 。 唯一的例外是 `amoptions` ，它应当被声明为接受 `text[]` 和 `bool` 并返回 `bytea` 。 这样就允许客户端代码执行 `amoptions` 以确认选项设置的有效性。

54.3. 索引扫描

在一个索引扫描里，索引访问方法负责把它拿到的那些据说匹配扫描键字的所有行之 TID 的回流。 访问方法不会卷入从索引的父表中实际抓取这些行的动作中，也不会判断他们是否通过了扫描的时间条件测试或者是其它条件。

一个扫描键字是形如 `_index_key_` `` _operator_``_constant_ `` 的 `WHERE` 子句的内部表现形式，这里的索引键字是索引中的一个字段，而操作符是和该索引字段相关联的操作符族的一个成员。 一个索引扫描拥有零个或者多个扫描键字，他们是隐含着 AND 的关系 —返回的行被认为是满所所有列出的条件的行。

访问方法可以声称自己是有损的或对特定的查询需要再检查。 这意味着，索引扫描会返回所有通过扫描键字的条目并可能加上一些不能通过的条目。 核心系统的索引扫描装置之后会对堆元组应用索引条件以确认是否真的这个元组应该被选中。 如果再检查选项没有被指定，索引扫描必须精确返回匹配的条目。

请注意，确保找到所有条目以及确保所有条目都通过给出的扫描键字的条件完全是访问方法的责任。 还有，核心系统将只是简单的把所有匹配扫描键字和操作符族的 `WHERE` 子句传递过来，而不会做任何语义分析，以判断他们是否冗余或者是否相互矛盾。 举例来说，给出 `WHERE x > 4 AND x > 14` where `x` ，这里的 `x` 是一个 b-tree 索引字段， 那么把第一个扫描键字识别成冗余的和可抛弃的工作是 b-tree `amrescan` 函数的事。 `amrescan` 过程中所需要的预处理的范围将由索引访问方法把扫描键字缩减为一个"规范化"形式的具体需要而定。

一些访问方法以明确的顺序返回索引项，另外一些则不是。 实际上访问方法可以支持两种不同的排序输出方法：

- 以自然的数据顺序返回条目的访问方法（比如btree）应该设置 `pg_am` . `amcanorder` 为真。 当前，这样的访问方法必须为它们的相等和排序操作符使用btree兼容的策略数。
- 支持排序操作的访问方法必须设置 `pg_am` . `amcanorderbyop` 为真。 这表明这个索引能够以满足 `ORDER BY` `` _index_key_``_operator_ `` `_constant_` 的顺序返回条目。 这种形式的扫描修饰子能够像前面描述的那样被传入 `amrescan` 。

`amgettuple` 函数有一个 `direction` 参数，它可以是 `ForwardScanDirection` (正常情况)或者 `BackwardScanDirection` 。 如果 `amrescan` 之后的第一次调用声明 `BackwardScanDirection` ，那么匹配条件的索引记录集是从后向前扫描的，而不是通常的从前向后扫描， 因此 `amgettuple` 必须返回索引中最后的匹配行，而不是通常情况下的第一条。 （这些事情只会在那些设置了 `amcanorder` 为真的访问方法上会发生。） 在第一次调用之后， `amgettuple` 必须准备从最近返回的条目的位置开始，在两个方向上进行扫描步进。 （但是，如果 `pg_am` . `amcanbackward` 为假，所有随后的调用将会和第一次调用使用相同的顺序。）

支持排序扫描的访问方法必须支持在扫描上"标记"位置，并在之后可以返回这个被标记的位置。 相同的位置可能被恢复多次。 然而，每次扫描只有一个位置需要被记住；一次新的 `ammarkpos` 调用会覆盖先前被标记的位置。 不支持排序扫描的访问方法仍然可以在 `pg_am` 中提供标记和恢复函数，但是如果被调用可以让这些函数抛出错误。

扫描位置和标记位置(如果存在)都必须在面对索引中存在并发插入和删除的时候保持一致性。 如果一条并发新插入的记录并未被一次扫描返回(而如果扫描开始的时候该记录就存在的话，则会被返回)， 或者说扫描通过重新扫描或者回头扫描返回这样的记录（即使它第一次跑的时候没有返回这样的行）， 对于系统来说，这种情况是可以接受的。 类似的还有，一个并发的删除可以反映，也可以不反应一个扫描的结果。 重要的是，插入或者删除不会导致扫描会略过或者重复返回本身不是被插入或者删除的条目。

如果索引存储原始的被索引数据值（而不是某种有损的表现形式）， 对支持index-only扫描是有用的， 这时，索引返回实际的数据而不仅是堆元组的TID。 只有在可见性映射显示这个TID在一个全可见页上是这才有效；否则还是必须检查堆元组的MVCC可见性。 但是访问方法不需要关心这件事。

作为 `amgettuple` 的替代，索引扫描可以可以通过一次 `amgetbitmap` 调用抓取所有元组来完成。 这明显可能比 `amgettuple` 更高效，因为可以避免在访问方法内部的加锁解锁。 一般而言， `amgetbitmap` 和重复调用 `amgettuple` 有相同的效果， 但是为了简化我们加入了一些限制。 首先， `amgetbitmap` 一次返回所有元组，并且不支持标记和恢复位置。 第二，在bitmap中返回的元组没有任何指定的顺序，这也是 `amgetbitmap` 没有 `direction` 参数的原因。 （排序操作也就永远不会被应用到这样的扫描上） 既然没有办法返回被索引元组的内容，也就不会有使用 `amgetbitmap` 的index-only扫描。 最后， `amgetbitmap` 不能保证对被返回元组上实施任何在Section 54.4中描述的锁。

注意，如果一个访问方法的内部实现不适合实现其中一个API的话， 允许一个访问方法只实现 `amgetbitmap` 和 `amgettuple` 中的一个。

54.4. 索引锁的考量

索引访问方法必须支持多个进程对索引的并发更新。 在索引扫描期间，PostgreSQL核心系统在索引上抓取 `AccessShareLock` ，并且在更新索引期间(包括 `VACUUM`)也会抓取 `RowExclusiveLock` 。 因为这些锁类型不会冲突，所以访问方法有责任处理任何它自己需要的更细致的锁需求。 把整个索引锁住的排他锁只是在创建和删除索引或者 `REINDEX` 的时候使用。

创建一个支持并发更新的索引类型通常要求对所需的行为进行广泛的并且细致的分析。 对于 b-tree 和 Hash 索引类型，你可以读取在 `src/backend/access/nbtree/README` 和 `src/backend/access/hash/README` 里面描述的设计决策。

除了索引自己内部的一致性要求之外，并发更新产生了一些有关父表(堆)和索引之间的一致性问题。 因为PostgreSQL是把堆的访问和更新与索引的访问和更新分开的。 用下面的规则处理这样的问题：

- 在制作一行的索引记录之前，先做堆记录。 （因此并发的索引扫描很可能看不到堆记录。 这么做应该是 OK 的，因为索引读者应该对未提交的行不感兴趣。 不过需要看看Section 54.5。）
- 如果一条堆记录要被删除(被 `VACUUM`)，所有其索引记录都必须首先删除。
- 对于并发索引类型，一次索引扫描必须在保存有 `amgettuple` 最后返回记录的索引页面上维护一个销，而 `ambulkdelete` 不能删除其它后端用销固定的索引页面里面的记录。 为何需要这条规则在下面解释。

如果没有第三条规则，那么一个索引读者是可能在一条索引记录刚要被 `VACUUM` 删除之前看到它， 然后在对应的堆记录已经被 `VACUUM` 删除时，去找这条堆记录。 如果读者到达该项时，该项编号仍然没有使用，那么这种情况不会导致严重的问题，因为空的项槽位会被 `heap_fetch()` 忽略。 但是如果第三个后端已经为其它什么东西复用了这个项槽位又如何? 如果使用 MVCC 兼容的快照，那么就不会有问题，因为新占据的槽位当然是太新了，因而无法通过快照测试。 但是，对于非 MVCC 兼容的快照(比如 `SnapshotNow`)，那么就有可能接受并返回一个实际上并不匹配扫描键字的行。 可以通过要求扫描键字在所有场合下都重新检查的方法来避免这种情况，但是这种方法开销太大了。 取而代之的是，通过在索引页面上使用一个销，当作一个代理，告诉系统说，读者可能还在对应堆记录的索引记录上空"飞行"。 让 `ambulkdelete` 在这样的销上阻塞可确保 `VACUUM` 无法在读者完成读取之前删除堆记录。 这种解觉办法只增加了一点点运行时开销，并且只是在非常罕见的实际有冲突的情况下才导致阻塞开销。

这个解决方法要求索引扫描必须是"同步的"：必须在扫描完对应的索引记录之后马上抓取每个堆记录。 这样的方案开销比较大，原因有若干个。 而"异步的"扫描，可以先从索引里收集很多 TID，然后在稍后的某个时间访问堆行，这样就会绕开很多索引锁的开销，以及可以允许

更有效的堆访问模式。 但是按照上面的分析，在非 MVCC 兼容快照上必须使用同步方法，而对使用 MVCC 快照的查询，使用异步扫描应该是可行的。

在 `amgetbitmap` 索引扫描里，访问方法不需要保证在任何返回的行上保持一个销。 毕竟，除了给最后一个行加销之外，也没法给其它的加。 因此，只能在 MVCC 兼容的快照里使用这样的扫描。

如果没有设置 `ampredlocks`，任何在可串行化事务中使用这个索引访问方法的扫描将会在整个索引上获得一个非阻塞的谓词锁。 与其并发的另一个可串行化事务向这个索引中插入任何一个元组时都会引发一个读写冲突。 如果在并发的可串行化事务间检测到某种模式的读写冲突，为了保证数据一致性其中一个事务可能会被取消。 如果设置了这个标志，表明这种索引访问方法实现了精细的谓词锁，因而趋向于削减这种事务取消的频度。

54.5. 索引唯一性检查

PostgreSQL使用唯一索引来强制 SQL 唯一约束，唯一索引实际上是不允许多条记录有相同键值的的索引。 一个支持这个特性的访问方法要设置`pg_am`.`amcanunique`为真。（目前，只有 b-tree 支持它。）

因为 MVCC，必须允许重复的条目物理上存在于索引之中：该条目可能指向某个逻辑行的后面的版本。 实际想强制的行为是，任何 MVCC 快照都不能包含两条相同的索引键字。 这种要求在向一个唯一索引插入新行的时候分解成下面的几种情况：

- 如果一个有冲突的合法行被当前事务删除，这是可以的。 （特别是因为一个 UPDATE 总是在插入新版本之前删除旧版本，这样就允许一个行上的 UPDATE 不用改变键字进行操作。）
- 如果一个在等待提交的事务插入了一行有冲突的数据，那么准备插入数据的事务必须等待看看改事务是否提交。 如果该事务回滚，那么就没有冲突。如果它没有删除冲突行然后提交，那么就有一个唯一性违例。 （实际上只是等待另外那个事务结束，然后在程序里重做可视性检查。）
- 类似的，如果一个有冲突的有效行被一个准备提交的事务删除，那么另外一个准备提交的插入事务必须等待该事务提交或者退出，然后重做测试。

此外，根据上面的规则报告唯一性违反前，访问方法必须重新检查刚被插入的行是否仍然"活着"。 如果这一行已经因为事务的提交而"死掉了"，那么不应当发出任何错误。 (这种情况不可能出现在插入一个由当前事务创建的行的普通场景下。 但是在`CREATE UNIQUE INDEX CONCURRENTLY`的过程中是可能的。）

我们要求索引访问方法自己进行这些测试，这就意味着它必须检查堆，以便查看那些根据索引内容表明有重复键字的任意行的提交状态。 这样做毫无疑问地很难看并且也不是模块化的，但是这样可以节约重复的工作： 如果我们实施分离的探测，那么，当查找新行的索引项的插入位置时，必须重复对冲突行的索引查找。 并且，没有很显然的方法来避免竞争条件，除非冲突检查是插入新索引项的整体动作的一部分。

如果唯一性约束是可延期的，情况将更加复杂：我们需要能够为新行插入一个新的索引项，但推迟任何唯一性违反的错误，直到语句结束甚至更晚。 为了避免不必要的重复搜索索引，索引访问方法应该在初始插入时做一个初步的唯一性约束检查。 如果结果明确地显示没有和活着的元组没有冲突，那么事情已经完成了。 否则当需要实施这个约束时，我们需要调度一个再检查。 如果再检查时，被插入的元组和有着相同键的其他元组都还活着，那么必须报告错误。 （为此，"活着"实际上意味着"在该索引项的HOT链上的任何一个元组还活着"。） 为了实现这个， `aminsert` 函数被传入拥有下列某一个值的`checkUnique`参数：

- `UNIQUE_CHECK_NO`指示不检查唯一性约束（这是一个非唯一索引）。

- `UNIQUE_CHECK_YES` 指示这是一个非可推迟唯一性约束，并且正如上面描述的必须立即检查唯一性约束。
- `UNIQUE_CHECK_PARTIAL` 指示这个唯一性约束是可延期的。 PostgreSQL将使用这种模式插入每一行的索引项。 访问方法必须允许在索引中插入重复的项目，并且通过让 `aminsert` 返回FALSE报告任何潜在的重复。 对每一个返回FALSE的行，一个延期的再检查将会被调度。

  访问方法必须识别任何可能违反唯一性约束的行，但是对它来说把不违反唯一性约束的行报告成可能违反并不是一个错误。 这允许不必等待其他事务完成就可以完成检查；在这里被报告的冲突不被当成错误并且将在以后进行再检查，那时冲突可能已经消失了。
- `UNIQUE_CHECK_EXISTING` 指示这是对一个被报告有潜在唯一性违反的行的被延期的再检查。 尽管是通过调用 `aminsert` 函数，这种情况下访问方法必须不能插入新的索引项。 相应的索引项已经存在了。当然，访问方法必须检查是否有另一个活着的索引项。 如果有并且目标行仍然活着的话报告错误。

  `UNIQUE_CHECK_EXISTING` 被调用时，建议访问方法进一步去确认目标行已经在索引中有一个索引项，如果没有则报错。 这是个好的做法，因为传入 `aminsert` 的索引元组的值可能被重新计算。 如果索引定义涉及不是真正不可变（immutable）的函数，我们可能会在索引中错误的区域查找。 检查目标行可以在再检查中被找到确保我们正在扫描原始插入时使用的相同的元组值。

54.6. 索引开销估计函数

系统给 `amcostestimate` 一些描述可能的索引扫描的信息，包括一个 WHERE 子句和 ORDER BY 子句的列表，这个子句列表是系统认为可以被索引使用的东西。 它必须返回访问该索引的开销估计值以及 WHERE 子句的选择性(也就是说，在索引扫描期间检索的将被返回的数据行在父表中所占据的比例)。 对于简单的场合，几乎开销估计器的所有工作都可以通过调用优化器里面的标准过程完成； 有 `amcostestimate` 这个函数的目的是允许索引访问方法提供和索引类型相关的知识，这样也许可以改进标准的开销估计。

每个 `amcostestimate` 函数都必须有下面这样的签名：

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

头3个参数是输入：

`root`

规划器的有关正在被处理的查询的信息。

`path`

被考虑的索引访问路径。除了开销和选择性其他所有的域都是有效的。

`loop_count`

索引扫描的重复次数，这个应该被考虑到开销估算中。 当考虑到用于嵌套循环内部的参数化扫描时，通常它大于1。 注意开销估算仍然只是为了一次扫描；一个大的 `loop_count` 的意思在多次扫描间进行某些缓存可能是合适的。

后面四个参数是传递引用的输出：

`*indexStartupCost`

设置为索引启动处理的开销

`*indexTotalCost`

设置为索引处理的总开销

`*indexSelectivity`

设置为索引的选择性

`*indexCorrelation`

设置为索引扫描顺序和下层的表的顺序之间的相关有效性

请注意开销估计函数必须用 C 写，而不能用 SQL 或者任何可用的存储过程语言，因为它们必须访问规划器/优化器的内部数据结构。

索引访问开销应该以 `src/backend/optimizer/path/costsize.c` 使用的单位进行计算：一个顺序磁盘块抓取开销是 `seq_page_cost` ， 一个非顺序抓取开销是 `random_page_cost` ， 而处理一个索引行的开销通常应该是 `cpu_index_tuple_cost` 。 另外，在任何索引处理期间调用的比较操作符，都应该增加一个数量为 `cpu_operator_cost` 倍数的开销(特别是计算索引条件 indexquals 自己的时候)。

访问开销应该包括所有与扫描索引本身相关的磁盘和 CPU 开销，但是不包括检索或者处理索引标识出来的父表的行的开销。

"启动开销"是总扫描开销中的这样一部分：在开始抓取第一行之前，必须花掉的开销。 对于大多数索引，这个可以是零，但是那些启动开销很大的索引类型可能不能把它设置为零。

`indexSelectivity` 应该设置成在索引扫描期间，父表中的行被选出来的部分的百分比。 在有损耗的查询的情况下，这个值通常比实际通过给出的查询条件的行所占的百分比要高。

`indexCorrelation` 应该设置成索引顺序和表顺序之间的相关性(范围在 -1.0 到 1.0 之间)。 这个数值用于调整从父表中抓取行的开销估计。

当 `loop_count` 大于1，返回的数值应当在每一次索引扫描之间平均。

开销估计

一个典型的开销估计器会像下面这样进行处理：

1. 基于给出的查询条件，估计并返回父表中将被访问的行的百分比。 如果缺乏索引类型相关的知识，那么使用标准的优化器函数 `clauselist_selectivity()`：

```
*indexSelectivity = clauselist_selectivity(root, path-&gt;indexquals,
                                           path-&gt;indexinfo-&gt;rel-&gt;relid,
                                           JOIN_INNER, NULL);
```

2. 估计在扫描过程中将被访问的索引行数。对于许多索引类型，这个等于 `indexSelectivity` 乘以索引中的行数，但是可能更多。 （请注意，索引的总页面数和行数可以从 `path->indexinfo` 结构中获得。）

3. 估计在扫描中将取出的索引页面数量。这个可能就是 `indexSelectivity` 乘以索引的总页面数。

4. 计算索引访问开销。一个通用的估计器可以这么干：

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they cost seq_page_cost each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
 */
cost_qual_eval(&index_qual_cost, path-&gt;indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

不过，上面没有考虑多次重复索引扫描中索引读取的开销分摊。

5. 估计索引的相关性。对于简单的在单个字段上的有序索引，这个值可以从 pg_statistic 中检索。 如果相关性是未知，那么保守的估计是零(没有相关性)。

开销估计器函数的例子可以在 `src/backend/utils/adt/selfuncs.c` 里面找到。

Chapter 55. GiST索引

Table of Contents

55.1. 介绍

GiST的意思是通用的搜索树(Generalized Search Tree)。 它是一种平衡树结构的访问方法，在系统中作为一个基本模版，可以使用它实现任意索引模式。 B-trees, R-trees和许多其它的索引模式都可以用GiST实现。

GiST的一个优点是，它允许伴随相应访问方法的定制数据类型由该数据类型领域里的专家而不是数据库专家来开发。

这里的有些信息是来自加州大学伯克力分校的 GiST 项目网站 和Marcel Kornacker的论文 Access Methods for Next-Generation Database Systems。 PostgreSQL里的GiST实现目前主要是 TeodorSigae 和 OlegBartunov 维护的，在他们的 网站上有更多信息。

55.2. 扩展性

传统上，实现一种新的索引访问方法意味着大量的艰苦工作。必须理解数据库的内部工作机制，比如锁的机制和预写日志。 GiST接口有一个高层的抽像，只要求访问方法的实现者实现被访问的数据类型的语意。 GiST层本身会处理并发，日志和搜索树结构等任务。

不要把这个扩展性和其它标准搜索树的扩展性混淆在一起，比如它们所能处理的数据等方面。 比如，PostgreSQL支持可扩展的 B-trees和哈希索引。 这就意味着可以用PostgreSQL在任意你需要的数据类型上建立 B-tree或哈希 。 但是 B-trees 只支持范围谓词(`<` 、 `=` 、 `>`),而哈希仅支持相等查询。

所以，如果你用PostgreSQL B-tree 索引了一个图像集，那么你就只能发出类似 "图像 x 和图像 y 相等吗"、"图像 x 是不是比图像 y 小"、"图像 x 是否大于图像 y"这样的查询。 依赖于你在这个环境下定义的"等于"、"小于"、"大于"的含义，上面这些查询可能有意义。 但是，使用一个基于GiST的索引，你可以创建一些方法来提出和领域相关的问题，比如"找出所有马的图像"或者"找出所有曝光过头的图像"。

要让一种GiST访问方法跑起来只要实现几个用户定义方法，这些方法定义了树里面的键字的行为。 当然，为了支持那些怪异的查询，这些方法也会相当怪异，但是对于所有标准的查询(B-tree，R-tree 等)，他们是相当直接的。 简单说，GiST组合了扩展性和通用性，以及代码复用和一个干净的界面。

GiST用的索引操作符类必须提供7个方法，第8个方法是可选的。 索引的正确性通过正确的实现 `same` ， `consistent` 和 `union` 方法来确保,而索引的效率(大小和速度)依赖于 `penalty` 和 `picksplit` 。 剩下的2个方法是 `compress` 和 `decompress` ， 它们允许索引持有的内部数据和它索引的对象数据的类型不同。 叶子节点的类型必须和被索引数据相同，而其他节点可以是任意C结构(但是，这里仍然必须遵守PostgreSQL中数据类型的规则， 对可变大小数据请参考 `varlena`)。 如果树的内部数据类型在SQL级别存在，可以在 `CREATE OPERATOR CLASS` 命令中使用 `STORAGE` 选项。 可选的第8个方法是 `distance` ,如果希望操作符类支持排序的扫描（最邻近搜索），就需要提供这个方法。

`consistent`

给定一个索引项 `p` 和查询值 `q` ， 这个函数决定是否索引项和查询"一致"； 也即是，对任何该索引项代表的行，谓词 "`` _indexed_column_``_indexable_operator_ `` `q` "是否可能为真？ 对叶子索引项这等价于测试索引条件，对内部树节点它指示是否有必要扫描该节点代表的索引子树。 当结果为 `true` ，必须还要返回 `recheck` 标志位。这指示了谓词是精确为真还是只是可能为真。 如果 `recheck` = `false` ，索引已经精确地测试了谓词条件， 如果 `recheck` = `true` ，相应的行仅仅是一个候选匹配。 这种情况下，系统还将自动对实际的行值进行评价 `_indexable_operator_` 以检查是否真的匹配。 这一规则允许GiST同时支持无损索引和有损索引。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_consistent(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry-&gt;key);
    bool        retval;

    /*
     * 根据strategy,key和query决定返回值。
     *
     * 使用GIST_LEAF(entry)可以感知函数在索引树的什么位置被调用，比如这在支持=操作符时很方便
     * （可以在非叶子节点检查非空的union()和在叶子节点检测等价性)。
     */

    *recheck = true;        /* 如果是精确检查则为假 */

    PG_RETURN_BOOL(retval);
}
```

这里 `key` 是索引中的一个元素，而 `query` 是要在索引中查找的值。 `StrategyNumber` 参数指示要应用操作符类中的哪个操作符， 它必须是 `CREATE OPERATOR CLASS` 命令指定的操作符编号之一。 依赖于在操作符类中包含的操作符， `query` 的数据类型可能和操作符不同，但是上面的骨架代码假设不是这种情况。

`union`

这个方法用于合并树中的信息。输入一个项目的集合，这个函数生成一个代表所有给定项目的新的索引项目。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_union(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec-&gt;vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec-&gt;n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i &lt; numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

正如你看到的，这个骨架代码中我们处理了符合`union(X, Y, Z) = union(union(X, Y), Z)`的数据类型。 在GiST支持方法中实现适当的union算法也可以很容易地支持其它不符合这一条件的数据类型。

`union`的实现函数应该返回一个由`palloc()`分配的内存的指针。 不能简单地直接返回输入的东西。

`compress`

把数据项转换为适合在索引页中存储的格式。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_compress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry-&gt;leafkey)
    {
        /* 把entry-&gt;key替换为压缩的版本 */
        compressed_data_type *compressed_data = palloc(sizeof(compressed_data_type));

        /* 从entry-&gt;key填充*compressed_data */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry-&gt;rel, entry-&gt;page, entry-&gt;offset, FALSE);
    }
    else
    {
        /* 通常不需要对非叶子节点做任何处理 */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

当然，为了压缩叶子节点，你需要把 `_compressed_data_type_` 适配到特定的数据类型。

根据你的需求，可能还需要关心如何压缩 `NULL` 值，例如存储为 `(Datum) 0` ，就像 `gist_circle_compress` 那样。

`decompress`

与 `compress` 函数正好相反。 把数据项的索引表现转换为可以被数据库处理的格式。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_decompress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

上面的骨架代码适合不需要解压缩的场合。

`penalty`

返回插入新项目到特定分支的"代价"值。 项目将会被插入到树中 `penalty` 最小的路径。 `penalty` 的返回值应该是非负数。 如果返回了负数将会被当作0处理。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- in some cases penalty functions need not be strict
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_penalty(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry-&gt;key);
    data_type  *new = DatumGetDataType(newentry-&gt;key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

`penalty` 函数对索引的性能非常重要。 在插入阶段，它可以用来决定把新增加项目插入到哪个分支。 在查询阶段，越平衡的索引，检索速度越快。

`picksplit`

如果需要分裂一个索引页面的时候，这个函数决定页面中哪些项目保存在旧页面里，哪些移动到新页面里。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_picksplit(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    OffsetNumber maxoff = entryvec-&gt;n - 1;
    GISTENTRY  *ent = entryvec-&gt;vector;
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    int         i,
                nbytes;
```

```
    OffsetNumber *left,
                 *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec-&gt;n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v-&gt;spl_left = (OffsetNumber *) palloc(nbytes);
    left = v-&gt;spl_left;
    v-&gt;spl_nleft = 0;

    v-&gt;spl_right = (OffsetNumber *) palloc(nbytes);
    right = v-&gt;spl_right;
    v-&gt;spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* 初始化项目数组 */
    raw_entryvec = (GISTENTRY **) malloc(entryvec-&gt;n * sizeof(void *));
    for (i = FirstOffsetNumber; i &lt;= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec-&gt;vector[i]);

    for (i = FirstOffsetNumber; i &lt;= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec-&gt;vector;

        tmp_union = DatumGetDataType(entryvec-&gt;vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * 选择放置索引项目的位置，并相应地更新unionL和unionR。
         * 追加项目到v_spl_left或者v_spl_right，并注意处理计数器。
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v-&gt;spl_nleft);
        }
        else
        {
            /*
             * 右边做相同处理
             */
        }
    }

    v-&gt;spl_ldatum = DataTypeGetDatum(unionL);
    v-&gt;spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

像 `penalty` 一样， `picksplit` 函数对索引的性能也非常重要， 设计合适的 `penalty` 和 `picksplit` 函数直接关系到实现良好性能的GiST索引。

`same`

2个索引项目等价时为真，否则为假。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_same(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_same(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

由于历史的原因， `same` 函数并不是单纯地返回布尔值， 而是将标志位存储到由第3个参数指向的位置。

`distance`

给定一个索引项目 `p` 和查询值 `q` ， 这个函数决定这2者之间的"距离"。 如果操作符类包含任何排序的操作符，必须要提供这个函数。 通过先返回最小"距离"值的索引项目，可以实现使用了排序操作符的查询，因此结果必须和操作符的语义一致。 对一个叶子索引项目，结果只是到索引项目的距离；对内部项目，结果必须是任何子节点项目的最小距离。

这个函数的SQL声明必须按照如下方式。

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

C模块中的对应代码可以参考下面的骨架代码。

```
Datum       my_distance(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    data_type  *key = DatumGetDataType(entry-&gt;key);
    double      retval;

    /*
     * determine return value as a function of strategy, key and query.
     */

    PG_RETURN_FLOAT8(retval);
}
```

`distance`函数的参数，除了recheck标志位，其他和`consistent`函数相同。 一个叶子节点的距离值必须是精确的，因为一旦返回了元组就没有办法再进行排序了。 对内部节点允许一定程度的近似，只要不大于任何一个子节点的实际距离。 比如，在地理应用中到矩形边界的距离就足够了。 结果值可以是任何有限的`float8`类型值。 （无穷和负无穷用于在内部作为空等情况使用，因此，不建议`distance`返回这些值。）

所有的GiST支持方法通常在短周期内存上下文中被调用，也就是说，在每个元组被处理后`CurrentMemoryContext`都会被重置。 因此不太需要担心pfree被palloc出来的所有东西。 然而，有些情况下，需要支持方法在多次调用间缓存数据。 为了实现这个目的，需要在`fcinfo->flinfo->fn_mcxt`中分配长生命周期的数据， 并且在`fcinfo->flinfo->fn_extra`中保存其指针。 这样的数据在索引操作(比如:单个的GiST索引扫描，索引创建或索引元组插入)完成后仍然有效。 在覆盖`fn_extra`的值前要小心的pfree掉先前的值，否则在操作期间内存泄漏会越积越多。

55.3. 实现

55.3.1. 缓存的GiST创建

通过插入所有元组来创建一个很大GiST索引通常速度会很慢。 因为如果索引元组被分散到索引的各个地方，并且索引因为太大而不能完全放入缓存，那么插入将需要触发很多随机IO。 从9.2版开始，PostgreSQL支持更加高效的基于缓存的GiST索引创建方法,这种方法可以极大地减少非排序数据集所需的随机IO。 但对于很好地排序过的数据集，这种方法效果甚微，甚至完全没有。 因为这种情况下一次只有少量的页接收新元组，即使整个索引不能放入缓存，这些页却可以完全放到缓存里。

但是，缓存的索引创建需要更多地调用 `penalty` 函数，这会消耗一些额外的CPU资源。 而且，缓存的索引创建使用的缓存需要占用临时的磁盘空间，最大为索引的最终大小。 缓存也可能影响最终生成的索引的质量，既有正面的也有负面的影响。 影响依赖于不同的因素，比如输入数据的分布和操作符类的实现。

缺省情况下，当索引大小达到effective_cache_size时，GiST索引创建会切换到缓存方式。 在执行CREATE INDEX命令时，也可以通过 `BUFFERING` 参数手动打开或关闭缓存方式。 缺省行为在大多数场合都是合适的，但是如果输入的数据是排过序的，关闭缓存可能多少会快一点。

55.4. 例

PostgreSQL的源码发布中包含了一些使用GiST实现的索引方法的例子。 当前的核心系统提供了全文检索支持(`tsvector` 和 `tsquery` 的索引)以及针对一些内建的几何数据类型(参阅 `src/backend/access/gist/gistproc.c`)的R-Tree等价的功能。 以下 `contrib` 模块也包含了GiST操作符类。

`btree_gist`

针对一些数据类型的B-tree等价功能

`cube`

多维立方体的索引

`hstore`

存储键值对的模块

`intarray`

int4值的一维数组的RD-Tree

`ltree`

树状结构的索引

`pg_trgm`

使用trigram匹配计算文本的相似度

`seg`

"浮点数范围"的索引

Chapter 56. SP-GiST索引

Table of Contents

56.1. 介绍

SP-GiST是空间分割的(Space-Partitioned)GiST的省略语。 SP-GiST支持分区的搜索树，这有助于开发四叉树，KD树，基数树(radix tree)等范围广泛的不同的非平衡数据结构。 这些结构的共通特征是它们反复地把搜索空间划分成大小不必相等的分区。能很好的匹配分区规则的查询会非常快。

这些流行的数据结构一开始是被设计用在内存中的。 在内存中，它们通常被设计为由指针链接起来的动态分配的节点的集合。 这不适合直接在磁盘上存储，因为这些指针链相当长可能需要太多的磁盘访问。 相对的，基于磁盘的数据结构应该在最小的I/O上有很高的展开。 SP-GiST要解决的挑战是把搜索树映射到磁盘页面上，通过这种方式即使遍历很多节点也只需访问很少的磁盘页面。

和GiST一样，SP-GiST允许伴随相应访问方法的定制数据类型由该数据类型领域里的专家而不是数据库专家来开发。

这里的有些信息是来自普渡大学(Purdue University)的 SP-GiST索引项目 网站。 PostgreSQL中的SP-GiST 实现目前主要是 Teodor Sigaev 和 Oleg Bartunov 维护的，在他们的 网站上有更多信息。

56.2. 扩展性

SP-GiST提供了一个高度抽象的接口,只需要访问方法开发人员实现特定于给定数据类型的方法。SP-GiST核心负责高效的磁盘映射和搜索树结构。它还负责考虑并发性和日志记录。

SP-GiST树的叶元组包含和被索引列相同数据类型的值。 在根层级的叶元组总是包含原来的索引数据值,但是低层级的叶元组可能只包含一个压缩表示,如一个后缀。 在这种情况下,操作符类支持函数必须能够，通过利用从到达叶元组所经过的内部元组中收集到的信息，重建原始值。

内部元组更为复杂,因为它们是搜索树中的分支点。 每个内部元组包含由一个或多个节点组成的集合,表示一组类似的叶元组值。 一个节点包含一个链接指向另一个低级内部元组,或指向一个短的叶元组的列表，这些叶元组存储在相同的索引页面中。 每个节点都有一个用来描述它的标签。例如,在一个基数树中节点标签可以是字符串值的下一个字符。 可选地,一个内部元组可以有一个前缀值,描述了它的所有成员。 在基数树中这可能是其所代表的字符串的共同前缀。 前缀值不一定真的是一个前缀,可以是操作符类所要求的任何数据。 例如,在四叉树中它可以存储可度量四个象限的中心点，四叉树的内部元组也就会相应的存储四个节点，每个代表了这个中心点周围的一个象限。

一些树算法需要知道当前元组的层级(或深度),所以SP-GiST核心为操作符类提供了在向下访问树时管理层级计数的能力。 并且也支持在需要时递增地重建所代表的值。

> **Note:** SP-GiST核心代码考虑了null条目。 尽管SP-GiST索引为被索引列中的null值存储了的条目,但这对索引操作符类代码是隐藏的:null索引条目或搜索条件不会被传递给操作符类的方法。 (假定SP-GiST操作符是严格的,因此不能在null值上匹配成功。)因此后面也就不会再讨论null值的问题了。

一个用于SP-GiST的索引操作符类必须提供五个用户定义的方法。 所有五个方法按照约定接受两个 `internal` 参数，第一个参数是一个指向包含了支持方法的输入值的C结构体的指针，而第二个参数是一个指向放置输出值的C结构体的指针。其中四个方法只是返回void,因为它们所有的结果都出现在输出结构体里;但 `leaf_consistent` 返回一个布尔结果。这些方法不能修改输入结构体中的任何域。在所有情况下，在调用用户定义的方法之前，输出结构体的内容被用初始化为0。

五个用户定义的方法如下：

`config`

返回关于索引实现的静态信息,包括前缀的数据类型OID和节点标签的数据类型。

函数的SQL声明必须看起来像这样:

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

第一个参数是一个指向C结构体 `spgConfigIn` 的指针,包含函数的输入数据。 第二个参数是一个指向C结构体 `spgConfigOut` 的指针,该函数必须填充结果数据到里面。

```
typedef struct spgConfigIn
{
    Oid         attType;        /* Data type to be indexed */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* Data type of inner-tuple prefixes */
    Oid         labelType;      /* Data type of inner-tuple node labels */
    bool        canReturnData;  /* Opclass can reconstruct original data */
    bool        longValuesOK;   /* Opclass can cope with values &gt; 1 page */
} spgConfigOut;
```

传递 `attType` 是为了支持多态索引操作符类。 对普通固定数据类型操作符类,它将总是有相同的值,因此可以忽略。

对于不使用前缀的操作符类, `prefixType` 可以被设置成 `VOIDOID` 。 同样,对不使用节点标签的操作符类， `labelType` 可以被设置成 `VOIDOID` 。 如果操作符类能够重建最初提供的索引值， `canReturnData` 应设置为true。 只有当 `attType` 是可变长类型并且操作符类能够通过反复的添加后缀分割很长的值的时候， `longValuesOK` 才应该被设置为true(参见Section 56.3.1)。

`choose`

选择一种方法将一个新值插入到一个内部元组。

函数的SQL声明必须看起来像这样:

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

第一个参数是一个指向C结构体 `spgChooseIn` 的指针,包含函数的输入数据。 第二个参数是一个指向C结构体 `spgChooseOut` 的指针,该函数必须填充结果数据到里面。

```
typedef struct spgChooseIn
{
    Datum       datum;          /* original datum to be indexed */
    Datum       leafDatum;      /* current datum to be stored at leaf */
    int         level;          /* current level (counting from zero) */

    /* Data from current inner tuple */
    bool        allTheSame;     /* tuple is marked all-the-same? */
    bool        hasPrefix;      /* tuple has a prefix? */
    Datum       prefixDatum;    /* if so, the prefix value */
    int         nNodes;         /* number of nodes in the inner tuple */
    Datum      *nodeLabels;     /* node label values (NULL if none) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* descend into existing node */
    spgAddNode,                 /* add a node to the inner tuple */
    spgSplitTuple               /* split inner tuple (change its prefix) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* action code, see above */
    union
    {
        struct                  /* results for spgMatchNode */
        {
            int         nodeN;      /* descend to this node (index from 0) */
            int         levelAdd;   /* increment level by this much */
            Datum       restDatum;  /* new leaf datum */
        }           matchNode;
        struct                  /* results for spgAddNode */
        {
            Datum       nodeLabel;  /* new node's label */
            int         nodeN;      /* where to insert it (index from 0) */
        }           addNode;
        struct                  /* results for spgSplitTuple */
        {
            /* Info to form new inner tuple with one node */
            bool        prefixHasPrefix;    /* tuple should have a prefix? */
            Datum       prefixPrefixDatum;  /* if so, its value */
            Datum       nodeLabel;          /* node's label */

            /* Info to form new lower-level inner tuple with all old nodes */
            bool        postfixHasPrefix;   /* tuple should have a prefix? */
            Datum       postfixPrefixDatum; /* if so, its value */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

`datum` 是被插入到索引的原始数据。 `leafDatum` 最初和 `datum` 是一样的，如果函数 `choose` 或者 `picksplit` 把它修改了，在树的低层级可能会不同。 当插入搜索到达叶页面，当前的 `leafDatum` 值就是存储到新生成的叶元组中的值。 `level` 是当前内部元组的层级，从0，也就是根的层级，开始计数。 如果当前内部元组包含多个等价节点， `allTheSame` 为true(参见Section 56.3.3)。 如果当前内部元组包含前缀， `hasPrefix` 为true。 此时， `prefixDatum` 是前缀值。 `nNodes` 是内部元组中包含子节点的数量， `nodeLabels` 是它们的标签值的数组,或者是NULL如果没有标签的话。

`choose` 函数可以确定新值匹配一个现有的子节点,或者必须添加一个新的子节点,或者新值与元组前缀不一致,所以内部元组必须分裂开以创建一个限制较少的前缀。

如果新值匹配的一个现有的子节点,把 `resultType` 设置为 `spgMatchNode` 。 把 `nodeN` 设置为那个索引节点在节点数组中的索引(从0开始)。 把 `levelAdd` 设置为，由下降到那个节点导致的 `level` 增量； 或者为0如果操作符类不使用层级。 把 `restDatum` 设置为和 `datum` 相等的值， 如果操作符类从一个层级到下一个层级不会修改数据值；否则将其设置为修改后的值，它在下一个层级被用作 `leafDatum` 。

如果必须添加一个新的子节点,把 `resultType` 设置为 `spgAddNode` 。 设置 `nodeLabel` 为新节点的标签,并设置 `nodeN` 为插入位置在节点数组中的索引(从0开始)。 添加了节点后, `choose` 函数将再次与被修改的内部元组一起被调用，那次调用应该导致一个 `spgMatchNode` 结果。

如果新值与元组前缀是不一致的,把 `resultType` 设置为 `spgSplitTuple` 。 这一动作把所有现有的节点移动到一个新的低层级的内部元组,并把现有内部元组替换为一个只有一个链接到新的低层级内部元组的单个节点的元组。 设定 `prefixHasPrefix` 表明是否新的较高的元组应该有一个前缀,如果是的话，设置 `prefixPrefixDatum` 为前缀值。 这个新的前缀值必须比原来的限制足够小以接受新的被索引值,而且应当不超过原前缀的长度。 设置 `nodeLabel` 为指向新的低层级内部元组的节点的标签值。 设定 `postfixHasPrefix` 表明是否新的较低的元组应该有一个前缀,如果是的话，设置 `postfixPrefixDatum` 为前缀值。 这两个前缀和额外的标签的组合必须与原始前缀具有相同的含义,因为没有机会改变被移动到新的低层级元组中的节点标签，也不能改变任何子索引条目。 节点被分裂后, `choose` 会被再次调用，针对替换的内部元组。那个调用通常会导致 `spgAddNode` 结果,因为分裂步骤添加的节点标签可能不会匹配新值；所以在那之后,还会有第三次调用,最后的调用返回 `spgMatchNode` ,允许插入操作下去到叶层级。

`picksplit`

决定如何在一组叶元组之上创建一个新的内部元组。

函数的SQL声明必须看起来像这样:

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

第一个参数是一个指向C结构体 `spgPickSplitIn` 的指针,包含函数的输入数据。 第二个参数是一个指向C结构体 `spgPickSplitOut` 的指针,该函数必须填充结果数据到里面。

```
typedef struct spgPickSplitIn
{
    int         nTuples;        /* number of leaf tuples */
    Datum      *datums;         /* their datums (array of length nTuples) */
    int         level;          /* current level (counting from zero) */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool        hasPrefix;      /* new inner tuple should have a prefix? */
    Datum       prefixDatum;    /* if so, its value */

    int         nNodes;         /* number of nodes for new inner tuple */
    Datum      *nodeLabels;     /* their labels (or NULL for no labels) */

    int        *mapTuplesToNodes;   /* node index for each leaf tuple */
    Datum      *leafTupleDatums;    /* datum to store in each new leaf tuple */
} spgPickSplitOut;
```

`nTuples` 是提供的叶元组的数量。 `datums` 是数据值的数组。 `level` 是所有的叶元组共享的当前层级,这将成为新的内部元组的层级。

`hasPrefix` 表明是否新的内部元组应该有一个前缀,如果是的话设置prefixDatum前缀值。 `nNodes` 表明新内部元组将包含的节点数量,并设置 `nodeLabels` 为它们的标签值的数组。 (如果节点不需要标签,设置 `nodeLabels` 为NULL;有关详细信息,请参见Section 56.3.2。) 设置 `mapTuplesToNodes` 为各个叶元组应该被分配的节点的索引(从0开始)的数组。 设置 `leafTupleDatums` 为存储在新的叶元组的值的数组(如果操作符类从一个层级到下一个层级不修改数据,它们将和输入数据相同)。 注意,`picksplit` 函数负责分配(palloc) `nodeLabels`, `mapTuplesToNodes` 和 `leafTupleDatums` 数组。

如果提供了不止一个叶元组,预计 `picksplit` 函数会把它们分类到多个节点,否则不可能把叶元组分裂到多个页面,这是这个操作的最终目的。 因此,如果 `picksplit` 函数最终把所有叶元组放在同一节点,核心SP-GiST代码将覆盖这一决定并生成一个内部元组, 这些叶元组会被随机分配到这个内部元组的几个有等价标签(identically-labeled)的节点上。 这样的元组被设置了 `allTheSame` 标志, 以表示发生这样的事情了。 `choose` 和 `inner_consistent` 函数必须小心对待这样的内部元组。 有关更多信息,请参见Section 56.3.3。

只有当 `config` 函数设置 `longValuesOK` 为true,并且提供了大于一页面的输入值时, `picksplit` 才可以被应用到单个叶元组。 在这种情况下操作的要点是剥离前缀并产生一个新的、更短的叶数据值。 这个调用将被反复执行, 直到叶数据已经短到可以放到一个被生成的页面上。 有关更多信息,请参见Section 56.3.1。

`inner_consistent`

返回树搜索需要继续访问的节点集合(分支)。

函数的SQL声明必须看起来像这样:

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

第一个参数是一个指向C结构体 `spgInnerConsistentIn` 的指针,包含函数的输入数据。 第二个参数是一个指向C结构体 `spgInnerConsistentOut` 的指针,该函数必须填充结果数据到里面。

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* array of operators and comparison values */
    int         nkeys;          /* length of array */

    Datum       reconstructedValue;     /* value reconstructed at parent */
    int         level;          /* current level (counting from zero) */
    bool        returnData;     /* original data must be returned? */

    /* Data from current inner tuple */
    bool        allTheSame;     /* tuple is marked all-the-same? */
    bool        hasPrefix;      /* tuple has a prefix? */
    Datum       prefixDatum;    /* if so, the prefix value */
    int         nNodes;         /* number of nodes in the inner tuple */
    Datum      *nodeLabels;     /* node label values (NULL if none) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* number of child nodes to be visited */
    int        *nodeNumbers;    /* their indexes in the node array */
    int        *levelAdds;      /* increment level by this much for each */
    Datum      *reconstructedValues;    /* associated reconstructed values */
} spgInnerConsistentOut;
```

长度为 `nkeys` 的 `scankeys` 数组描述了索引搜索条件。 这些条件以"AND"联接在一起，即，只有满足所有条件的索引条目才能匹配这个查询。 (注意,如果 `nkeys` 为0意味着所有索引条目都满足查询。) 通常这个函数只关心每个数组项目的 `sk_strategy` 和 `sk_argument` 字段,它们分别给出了可索引的操作符和比较值。 特别是没有必要看 `sk_flags` 以检查比较值是否为NULL,因为SP-GiST核心代码会过滤掉这样的条件。 `reconstructedValue` 是为父元组重建的值；如果在根层级或 `inner_consistent` 函数在父层级没有提供一个值，它会是0。 `level` 是当前内部元组的层级，从0，也就是根的层级，开始计数。 `returnData` 为 `true` ，如果这个查询需要重建数据的话；只有 `config` 函数声明了 `canReturnData` 时，才有可能是这样。 `allTheSame` 是true，如果当前内部元组标记"all-the-same";在这种情况下的所有节点具有相同的标签(如果有的话),所以要么全部要么没有一个匹配这个查询(参见Section 56.3.3)。 如果当前内部元组包含前缀， `hasPrefix` 为true。 此时， `prefixDatum` 是前缀值。 `nNodes` 是内部元组中包含子节点的数量， `nodeLabels` 是它们的标签值的数组,或者是NULL如果节点没有标签。

`nNodes` 必须被设置为搜索需要访问的子节点的数量, 并且 `nodeNumbers` 必须被设置的它们的索引的数组。 如果操作符类跟踪层级,设置 `levelAdds` 为向下访问到每个节点时层级增量的数组。 (通常这些增量对所有节点是相同的,但这并不一定是这样,所以使用一个数组)。 如果需要重建数据值,设置 `reconstructedValues` 为每个要访问的子节点的重建值的数组;否则,保持 `reconstructedValues` 为NULL。 注意, `inner_consistent` 函数负责分配(palloc) `nodeNumbers` , `levelAdds` 和 `reconstructedValues` 数组。

`leaf_consistent`

如果叶元组满足查询返回true。

函数的SQL声明必须看起来像这样:

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

第一个参数是一个指向C结构体 `spgLeafConsistentIn` 的指针,包含函数的输入数据。 第二个参数是一个指向C结构体 `spgLeafConsistentOut` 的指针,该函数必须填充结果数据到里面。

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* array of operators and comparison values */
    int         nkeys;          /* length of array */

    Datum       reconstructedValue;     /* value reconstructed at parent */
    int         level;          /* current level (counting from zero) */
    bool        returnData;     /* original data must be returned? */

    Datum       leafDatum;      /* datum in leaf tuple */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* reconstructed original data, if any */
    bool        recheck;        /* set true if operator must be rechecked */
} spgLeafConsistentOut;
```

长度为 `nkeys` 的 `scankeys` 数组描述了索引搜索条件。 这些条件以"AND"联接在一起，即，只有满足所有条件的索引条目才能匹配这个查询。 (注意,如果 `nkeys` 为0意味着所有索引条目都满足查询。) 通常这个函数只关心每个数组项目的 `sk_strategy` 和 `sk_argument` 字段,它们分别给出了可索引的操作符和比较值。 特别是没有必要看 `sk_flags` 以检查比较值是否为NULL,因为SP-GiST核心代码会过滤掉这样的条件。 `reconstructedValue` 是父元组重建的值；如果在根层级或 `inner_consistent` 函数在父层级没有提供一个值，它会是0。 `level` 是当前叶元组的层级，从0，也就是根的层级，开始计数。 `returnData` 为 `true` ，如果这个查询需要重建数据的话；只有 `config` 函数声明了 `canReturnData` 时，才有可能是这样。 `leafDatum` 是存储在当前叶元组中的键值。

如果这个叶元组匹配查询，这个函数必须返回 `true` ,否则返回 `false` 。 在 `true` 的情况下,如果 `returnData` 是 `true` ， 那么 `leafValue` 必须被设置为最初提供的这个叶元组索引的值。 如果匹配是不确定的, `recheck` 也被设置为 `true` ，因此操作符必须被重新应用到实际的堆元组上以验证匹配。

所有SP-GiST支持方法通常在一个短期的内存上下文中被调用;也就是说, 在处理每一个元组后 `CurrentMemoryContext` 将被重置。 因此不太需要担心地去pfree你palloc出来的所有东西。 （ `config` 方法是一个例外:它应该尽量避免内存泄露。 但通常 `config` 方法除了把常数赋值到传递过来的结构体中外，其它什么也不需要做。）

如果被索引列是一个collatable数据类型,该索引排序规则将被传递给所有的支持方法,使用标准的 `PG_GET_COLLATION()` 机制。

56.3. 实现

本节讨论实现细节和其他对SP-GiST操作符类的实现者有用的技巧。

56.3.1. SP-GiST的限制

单个的叶元组和内部元组必须容纳在一个索引页(默认8KB)里。 因此,当索引可变长数据类型的值时,长值只能被像基数树这样的方法支持,在树的每一层包含一个前缀,这个前缀足够短,可容纳在一个页面上,并且最后的叶子层级包括的后缀也足够短以容纳在一个页面上。 操作符类只有准备好了处理这样的事情，才应该设置 `longValuesOK` 为TRUE。 否则,SP-GiST核心将拒绝一个太大以致不能装入一个页面的数据值的索引请求。

同样,不要让内部元组增长得过大以致不能容纳在一个索引页面,这也是操作符类的责任；这限制了在一个内部元组里可以使用的子节点的数量,以及一个前缀值的最大大小。

另一个限制是,当一个内部元组的节点指向一组叶元组,这些元组必须都在相同的索引页面上。(这个设计决定是为了减少寻址以及节省把这些元组联接在一起的链接的空间)。 如果叶元组的集合增大到超过一个页面,就会执行分裂并插入一个中间的内部元组。 为了解决这个问题,新内部元组必须把叶子中的值的集合划分到多个节点组。 如果操作符类的 `picksplit` 函数未能这样做,SP-GiST 将会采取Section 56.3.3中描述的非常手段。

56.3.2. 没有节点标签的SP-GiST

一些树算法在每个内部元组中使用固定数量的节点; 例如,在四叉树中总是正好的有4个节点,它们代表围绕内部元组的中心点的四个象限。 在这种情况下代码典型的通过数值和节点打交道,没有必要有显式的节点标签。 为了废止节点标签(从而节省一些空间), `picksplit` 函数可以为 `nodeLabels` 数组返回NULL。 结果这将导致随后调用 `choose` 和 `inner_consistent` 函数时， `nodeLabels` 为NULL。 原则上,节点标签可用于一些内部元组，而在相同索引的其它元组上省略。

当使用一个包含无标签节点的内部元组时, `choose` 函数返回 `spgAddNode` 是错误的, 因为在这种情况下，节点的集合被假定为固定的。 同样也没有在 `spgSplitTuple` 动作中产生无标签节点的条款，因为预计将需要一个 `spgAddNode` 动作。

56.3.3. "All-the-same"内部元组

如果操作符类的 `picksplit` 函数不能把提供的叶子值划分到至少两个分类中， SP-GiST核心可以覆盖 `picksplit` 函数的结果。 如果发生了这种事情,包含多个节点的新的内部元组会被创建。 每个节点的标签都和 `picksplit` 给一个节点使用的标签相同(如果有的话),并且叶值被随

机划分到这些等价节点中。 `allTheSame` 标志会被设置到这个内部元组上，以警告 `choose` 和 `inner_consistent` 函数这个元组可能没有它们期望的节点集合。

当处理一个 `allTheSame` 元组, `choose` 的结果 `spgMatchNode` 被解释为新的值可以被分配到任何等价节点;核心代码将忽略提供的 `nodeN` 值并随机的进入其中一个节点(以保持树的平衡)。 `choose` 返回 `spgAddNode` 是错误的，因为将使这些节点不再是所有都等价。 如果要插入的值不匹配现有的节点，必须使用 `spgSplitTuple` 动作。

当处理一个 `allTheSame` 元组, `inner_consistent` 函数应该返回所有或没有一个节点作为继续索引搜索的目标,因为他们都是等价的。 这可能需要也可能不需要任何特殊的代码,这取决于 `inner_consistent` 函数通常假设了多少节点的含义。

56.4. 例

PostgreSQL源码发布包括几个SP-GiST索引操作符类的例子。 目前核心系统提供了文本列上的基数树和点的两种树类型:四叉树和KD树。 如果要查看代码，可进入 `src/backend/access/spgist/` 。

Chapter 57. GIN索引

Table of Contents

57.1. 介绍

GIN的意思是通用倒排索引(Generalized Inverted Index)。 GIN被设计用于这样一种情况：被索引项是组合值， 而被索引处理的查询需要搜索出现在这些组合值中的元素值。 比如，项目可能是文档，查询可以是搜索包含多个特定单词的文档。

我们使用项目这个词指代被索引的组合值，键指代一个元素值。 GIN本身总是存储和搜索键，而不是项目值。

GIN索引存储一系列(key, posting list)对， 这里的*posting list*是一组出现键的行ID。 每一个被索引的项目都可能包含多个键，因此同一个行ID可能会出现在多个posting list中。 每个键值只被存储一次，因此在相同的键出现在很多项目的情况下，GIN索引是非常紧凑的。

GIN索引之所以是通用的，是因为GIN 访问方法不需要了解它所加速的操作。 取而代之的是，它使用为特殊数据类型定义的定制策略。 策略定义了如何从被索引项目和查询条件中抽出键，以及如何决定包含了查询中一些键的一行是否确实满足查询条件。

GIN的一个优点是它允许由领域专家而不是由数据库专家来 开发附带适当的访问方法的自定义数据类型。 这一点与使用GiST很相似。

PostgreSQL中的GIN 实现主要由 Teodor Sigaev 和 Oleg Bartunov 维护。 关于GIN的更多信息可以访问他们的 网站。

57.2. 扩展性

GIN接口有一个高层次的抽象，仅要求实现被访问数据类型的语义即可。 GIN层自身可以处理并发操作、记录日志、搜索树结构。

定义一个GIN访问方法所要做的所有事情就是实现四个(或五个)用户定义的方法， 这些方法定义了键在树中的行为、键与键之间的关系、被索引的项目、能够使用索引的查询。 简而言之，GIN将扩展性与普遍性、代码重用、清晰的接口结合在了一起。

一个GIN索引操作符类必须实现的四个方法如下：

`int compare(Datum a, Datum b)`

比较两个键(不是被索引的项目!)然后返回一个小于、等于或大于零的值，分别表示第一个键小于、等于或大于第二个键。 NULL的键永远不会被传入这个函数。

`Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)`

给定一个被索引的项目，返回一个对应的由palloc分配的键的数组。返回的键的数目必须存储在 `*nkeys` 中。 如果任何键可能为NULL，还要palloc一个包含 `*nkeys` 个 `bool` 元素的数组，将地址存储到 `*nullFlags` ， 并且根据需要设置NULL值。 如果所有键都是非NULL的，可以让 `*nullFlags` 保持为 `NULL` （它的初始值）。 如果输入的项目不包含任何键，返回值可以为 `NULL` 。

`Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **`

给定一个被查询的值，返回一个对应的palloc分配的键数组。 也就是说， `query` 是可索引操作符右侧的值，而该操作符左侧是被索引的字段。 `n` 是操作符类中的操作符策略号(参见Section 35.14.2)。 通常， `extractQuery` 需要考量 `n` 来决定 `query` 的数据类型以及提取键值的方法。 返回的数组的元素个数必须存放在 `*nkeys` 中。 如果任何键可能为NULL，还要palloc一个包含 `*nkeys` 个 `bool` 元素的数组， 将地址存储到 `*nullFlags` ， 并且根据需要设置NULL值。 如果所有键都是非NULL的，可以让 `*nullFlags` 保持为 `NULL` （它的初始值）。 如果 `query` 不包含任何键，返回值可以为 `NULL` 。

`searchMode` 是一个输出参数，它允许 `extractQuery` 指定一些关于如何执行搜索的细节。 如果 `*searchMode` 被设置成 `GIN_SEARCH_MODE_DEFAULT` (这也是调用函数前它被初始化的值)， 只有匹配至少一个返回的键才能被认为是候选的匹配。 如果 `*searchMode` 被设置成 `GIN_SEARCH_MODE_INCLUDE_EMPTY` ， 除了包含至少一个匹配的键的项目，根本不包含任何键的项目也被视为候选的匹配。 （这个模式对于实现像"是否是子集"这样的操作是有用的） 如果 `*searchMode` 被设置成 `GIN_SEARCH_MODE_ALL` ， 索引中所有非NULL的项目都被认为是候选的匹配，不管它们是否匹配返回的键中的任何一个。 （这个模式比起其它两个要慢很多，因

为它必须要扫描整个索引，但这对正确的实现边界条件可能是必要的。 一个需要这种模式的操作符在大多数时候很可能不是一个好的GIN操作符类的候选。） 用于设置这个模式的符号定义在 `access/gin.h` 中。

`pmatch` 是在部分匹配时需要用到的一个输出参数。 如果使用它， `extractQuery` 必须分配一个有 `*nkeys` 个布尔值的数组， 并把数组地址保存到 `*pmatch` 。 数组的每个元素应该被设置为：TRUE，如果相应的键需要部分匹配；或者FALSE，如果不是。 如果 `*pmatch` 被设置为 `NULL` ，GIN假设不需要部分匹配。 在函数调用前这个值被初始化成了 `NULL` ， 因此，对于不支持部分匹配的操作符类，可以简单的忽略这个参数。

`extra_data` 是一个允许 `extractQuery` 传递额外数据给 `consistent` 和 `comparePartial` 的输出参数。 如果使用它， `extractQuery` 必须分配一个包含 `*nkeys` 个Pointer元素的数组， 并把数组地址保存到 `*extra_data` ，然后把它想附加的东西存储到各个独立的指针中。 在函数调用前这个值被初始化成了 `NULL` ， 因此，对于不需要附加数据的操作符类，可以简单的忽略这个参数。 如果 `*extra_data` 被设置了，那么整个数组会被传给 `consistent` 方法， 适当的元素会被传给 `comparePartial` 方法。

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_da
```

如果被索引项目满足策略号为 `n` 的查询操作符（或可能满足，如果recheck指示符被返回了的话）返回TRUE。 这个函数并不直接访问被索引项目的值，因为GIN并没有精确的把项目保存下来， 但是需要知道哪些从查询中提取的键值出现在给定的索引项目中。 `check` 数组的长度是 `nkeys` ，这与先前针对这个 `query` 调用的 `extractQuery` 函数返回的键值的数目相同。 如果被索引项目包含了相应的查询键， `check` 数组中对应的元素值就是TRUE。 比如，如果(check[i] == TRUE)，那么意味着 `extractQuery` 的结果数组的第i个键出现在了索引项目中。 考虑到 `consistent` 可能会用到，原始的 `query` 也被作为参数传入进来。 与此相同的还有 `extractQuery` 函数返回的 `queryKeys[]` 和 `nullFlags[]` 。 `extra_data` 是 `extractQuery` 函数返回的额外数据数组，如果没有的话就是 `NULL` 。

当 `extractQuery` 在 `queryKeys[]` 中返回一个NULL的键值， 如果被索引项目包含NULL键值，相应的 `check[]` 中的元素是TRUE。 也就是说， `check[]` 的语义很像 `IS NOT DISTINCT FROM` 。 如果需要知道是通常值匹配还是NULL匹配， `consistent` 函数可以检查相应的 `nullFlags[]` 元素。

成功执行后，如果对这个元组需要执行查询操作符是否匹配的再检查， `*recheck` 需要被设置为TRUE， 如果索引测试已经是精确的了，则设为FALSE。 也就是说，FALSE的返回值确保堆元组不匹配这个查询； 伴随 `*recheck` 为FASLE的TRUE的返回值确保堆元组匹配这个查询； 伴随 `*recheck` 为TRUE的TRUE的返回值意味着堆元组可能匹配这个查询， 因此需要取得这个堆元组，并通过直接针对原始的被索引项目评估查询操作符的方式进行再检查。

GIN操作符类可以可选地提供第五个函数。

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)
```

比较一个部分匹配查询键和一个索引键。 返回一个整形值，它个符号代表了不同的含义：小于0意味着索引键不匹配查询，但是索引扫描应该继续； 0意味着索引键匹配查询；大于0指示应该终止索引扫描，因为不可能再有更多的匹配。 这里提供了生成部分一致查询的操作符的策略号 `n` ，以防需要用它的语义去决定何时终止扫描。 同样的， `extra_data` 是 `extractQuery` 生成的额外数据数组中的相应元素，或者为NULL，如果没有的话。 NULL的键永远不会被传入这个函数。

为了支持"部分匹配"查询，一个操作符类必须提供 `comparePartial` 方法， 并且当遇到部分匹配查询时，它的 `extractQuery` 方法必须设置 `pmatch` 参数。 详细请参考Section 57.3.2。

上面的各种 `Datum` 值的实际数据类型根据操作符类的不同而不同。 传入到 `extractValue` 中的项目值总是操作符类的输入类型， 所有的键值类型必须这个类的 `STORAGE` 类型。 传入到 `extractQuery` 和 `consistent` 的 `query` 参数的类型 是由策略号识别的类成员操作符的右操作数的输入类型。 它不需要和项目类型相同，只要可以从中抽取出正确类型的键值。

57.3. 实现

在内部，GIN索引包含一个在键上构造的B-tree索引， 每个键是一个或多个被索引项目的一个元素(比如，一个数组的一个成员)。 并且叶子页上每个元组包含了或者堆指针的B-tree的一个指针（一个"posting tree"）， 或者，当列表小到足以和键值一起存储到一个索引元组中时，则是堆指针的一个简单列表（一个"posting list"），

从PostgreSQL 9.1开始，NULL的键值可以被包含到索引里。 对NULL的或根据 `extractValue` 不包含任何键的被索引项，占位符null被包含到了索引中。 这就允许应该找到空项目的搜索可以执行。

多列GIN索引通过在组合值（列号，键值）上建立一个单个的B-tree实现。 不同列的键值可以有不同的类型。

57.3.1. GIN快速更新技术

由于倒排索引的本质特性，更新一个GIN索引可能会比较慢。 插入或更新一个堆行可能导致许多往索引的插入（从被索引项目中抽取出的每一个键一个）。 从PostgreSQL 8.4开始，GIN可以通过插入新的元组到一个临时的，待处理实体的未排序列表，来推迟很多这样的工作。 当表被vacuumed，或者如果待处理实体的列表太大了（大于work_mem）， 这些实体被使用和初始索引创建时用到的相同的bulk插入方法，移动到主要的GIN数据结构。 即使把额外的vacuum开销算进去，这也大大提升了GIN索引更新的速度。 而且，这种额外开销的工作可以通过后台进程而不是前端查询来处理。

这种方法的主要缺点在于搜索时除了常规的索引还必须要扫描待处理实体的列表。 因此，大的待处理实体的列表会显著的拖慢搜索。 另一个缺点是，虽然大多数更新很快，一个导致待处理列表(pending list)变得"太大"的更新将引发一个立即的清理周期， 并因此比起其它更新会非常慢。 恰当的使用autovacuum可以最小化这两个问题。

如果一致的响应时间比更新速度更重要，可以通过把GIN 索引的存储参数 `FASTUPDATE` 设置为off而不使用待处理实体。 详细请参考CREATE INDEX。

57.3.2. 部分匹配算法

GIN可以支持"部分匹配"查询。即：查询并不决定单个或多个键的一个精确的匹配， 而是，可能的匹配落在一个合理的狭窄键值范围内（根据 `compare` 支持函数决定的键值排序顺序）。 此时， `extractQuery` 方法并不返回一个用于精确匹配的键值，取而代之的是， 返回一个要被

搜索的键值范围的下边界，并且设置`pmatch`为true。 然后，这个键值范围被使用`comparePartial`进行扫描。 `comparePartial`必须为一个相匹配的索引键返回0，不匹配但依然在被搜索范围内时返回小于0的值，对超过可以匹配的范围的索引键则返回大于0的值。

57.4. GIN提示与技巧

创建 vs 插入

由于可能要为每个项目插入很多键，所以GIN索引的插入可能比较慢。 对于向表中大量插入的操作，我们建议先删除GIN索引，在完成插入之后再重建它。

由于从PostgreSQL 8.4开始可以使用延迟索引了，这个建议已经没有那么必要了。（详细请参考Section 57.3.1。） 但是，对于非常大量的更新，最好还是先删除，而后再重建索引。

maintenance_work_mem

GIN索引的构建时间对 `maintenance_work_mem` 的设置非常敏感。 它没有为在索引创建期间少使用工作内存做出努力。

work_mem

在一系列往已有的启用了 `FASTUPDATE` 的GIN索引的插入操作期间， 只要待处理实体列表的大小超过了 `work_mem` ，系统就会清理这个列表。 为了避免可观察到的响应时间的大起大落，让待处理实体列表在后台被清理是比较合适的（比如通过autovacuum）。 前台清理操作可以通过增加 `work_mem` 或者更加激进的autovacuum来避免。 然而，扩大 `work_mem` 意味着如果发生了前台清理，那么它的执行时间将更长。

gin_fuzzy_search_limit

开发GIN索引的主要目的是为了让PostgreSQL 支持高度可伸缩的全文索引，并且常常会遇见全文索引返回海量结果的情形。 而且，这经常发生在查询高频词的时候，因而得到的结果集没什么用处。 因为从磁盘读取大量记录并对其进行排序会消耗大量资源，这在产品环境下是不能接受的(注意，索引搜索本身是很快的)。

为了易于控制这种情况，GIN有一个可配置的返回结果行数的软上限配置参数 `gin_fuzzy_search_limit` 。 缺省值 0 表示没有限制。如果设置了非零值，那么返回的结果就是从完整结果集中随机选择的一部分。

"软"的意思是实际返回的结果集大小可能与指定值稍有出入，具体取决于查询以及系统的随机数发生器的品质。

经验上，数千的值(比如5000 — 20000)可以工作得很好。

57.5. 限制

GIN假定可索引的操作符是严格的。 也就是说，对于NULL项目值， `extractValue` 根本不会被调用（取而代之是一个被自动创建的索引实体占位符）；并且，对于NULL查询， `extractQuery` 也根本不会被调用（取而代之是这样的查询被假定为不满足条件）。 不过，包含在非NULL复合项目或查询值中的NULL键值是支持的。

57.6. 例子

PostgreSQL的源码发布中包含了 `tsvector` 和 所有内部类型的一维数组的GIN操作符类。 `tsvector` 上的前缀搜索利用GIN的部分匹配特性实现。 以下 `contrib` 模块也包含了GIN操作符类。

`btree_gin`

许多数据类型的B-tree等价功能

`hstore`

存储键值对的模块

`intarray`

对 `int[]` 的增强支持

`pg_trgm`

使用三连词（trigram）匹配的文本相似度计算

Chapter 58. 数据库物理存储

Table of Contents

本章对PostgreSQL数据库使用的物理存储 格式提供一个概述。

58.1. 数据库文件布局

本节在文件和目录的层次上描述存储格式。

传统上，数据库集群所需要的配置和数据文件都存储在集群的数据目录里， 通常用环境变量 `PGDATA` 来引用。（用于定义它的环境变量名称之后） `PGDATA` 的一个常见位值 `/var/lib/pgsql/data` 。 不同服务器实例管理的多个集群， 可以在同一台机器上共存。

`PGDATA` 目录包含一些子目录和控制文件，在Table 58-1中显示。 除了这些必要的东西外，集群配置文件 `postgresql.conf` ， `pg_hba.conf` 和 `pg_ident.conf` 通常都存储在 `PGDATA` 这里。(尽管PostgreSQL 8.0和之后版本中，有可能把它们放在其他地方）。

Table 58-1. `PGDATA` 内容

项	描述
`PG_VERSION`	一个包含PostgreSQL主版本号的文件
`base`	包含与每个数据库对应的子目录的子目录
`global`	包含集群范围的表的子目录，比如 `pg_database`
`pg_clog`	包含事务提交状态数据的子目录
`pg_multixact`	包含多重事务状态数据的子目录(使用共享的行锁)
`pg_notify`	包含LISTEN/NOTIFY状态数据的子目录
`pg_serial`	包含已提交可串行化事务信息的子目录
`pg_snapshots`	包含输出快照的子目录
`pg_stat_tmp`	包含临时文件的统计子系统的子目录
`pg_subtrans`	包含子事务状态数据的子目录
`pg_tblspc`	包含指向表空间的符号链接的子目录
`pg_twophase`	包含用于预备事务的状态文件的子目录
`pg_xlog`	包含WAL(预写日志)文件的子目录
`postmaster.opts`	一个记录服务器最后一次启动时使用的命令行参数的文件
`postmaster.pid`	一个锁文件， 记录着当前服务器主进程ID(PID)，集群数据目录路径，服务器主起始时间戳，端口号， Unix-域套接目录路径（Windows上空），第一个有效listen_address(IP地址或者 `*` ， 如果不监听TCP，则为空)，以及共享内存段ID， （在服务器关闭之后此文件就不存在了）。

对于集群里的每个数据库，在 `PGDATA``/base` 里都有对应的一个子目录， 子目录的名字是该数据库在 `pg_database` 里的OID。 这个子目录是该数据库文件的缺省位置；特别值得一提的是， 该数据库的系统表存储在此。

每个表和索引都存储在独立的文件里，对于普通关系，这些文件 以该表或者该索引的*filenode*号命名， 该号码可以在 `pg_class` . `relfilenode` 中找到。 但是对于临时性关系，文件名称形式 `t``_BBB_` `_*FFF* ， 其中 *BBB* `是创建文件的后端ID，并且` *FFF* `是filenode号。 在任何情况下，除了主文件` *fsm*`的文件里。 表也有可见映射_，存储在一个分叉文件，后缀为 `_vm` ， 用来跟踪那些已知没有死行的页， 该可见映射在Section 58.4进一步的描述。 未记录的表和索引有三分之一分支，被称之为初始化分支，使用后缀 `_init` 存储在分支中（参阅Section 58.5）。

Caution

请注意，虽然一个表的filenode通常和它的OID相同，但实际上并不必须如此； 有些操作，比如 `TRUNCATE` ，`REINDEX` ，`CLUSTER` 以及一些特殊的 `ALTER TABLE` 形式， 都可以改变filenode而同时保留OID。避免假定filenode和表OID相同。 还有，对于某种系统表包括 `pg_class` 自身， `pg_class` . `relfilenode` 包含零。 这些表的实际的filenode编号存储在低级别的数据结构， 并且可以使用 `pg_relation_filenode()` 函数获取。

在表或者索引超过1 GB之后，将分割成1GB大小的段。 第一个段的文件名和filenode相同；随后的段名名为filenode.1, filenode.2 ... 等等。 这样的安排避免了在某些平台上的有文件大小限制的问题。 （实际上，1GB只是缺省的段大小。当构建PostgreSQL时， 可以使用配置选项 `--with-segsize` 调整段大小。） 原则上，自由空间映射和可见映射叉文件可能需要多个段， 尽管这在实践中不可能发生。

一个表如果有些字段里面可能存储相当大的数据， 那么就会有个相关联的*TOAST*表， 用于存储无法在表的数据行中放置的超大行外数据。 如果有的话， `pg_class` . `reltoastrelid` 从一个表链接到它的TOAST表。 参阅Section 58.2获取更多信息。

表的内容和索引在Section 58.6中有讨论。

表空间把情况搞得更复杂些。 每个用户定义的表空间都在 `PGDATA``/pg_tblspc` 目录里面有一个符号连接， 它指向物理的表空间目录(就是在 `CREATE TABLESPACE` 命令里声明的那个目录)。 这个符号连接是用表空间的 OID 命名的。 在物理的表空间目录内部，有个依赖PostgreSQL服务器版本的命名的子目录， 如 `PG_9.0_201008051` 。 （使用这个子目录的原因是为了让后续版本的数据库不产生冲突的情况下， 可以使用相同的 `CREATE TABLESPACE` 位置值。） 在有指定版本的子目录里，每个在表空间中有元素的数据库有个子目录， 命名为数据库的OID。表和索引存储在那个目录， 使用filenode命名方法。 `pg_default` 没有通过 `pg_tblspc` 关联， 但是对应 `PGDATA``/base` 。 类似的还有， `pg_global` 没有通过 `pg_tblspc` 关联， 而是对应 `PGDATA``/global` 。

`pg_relation_filepath()` 函数用于显示任何关系的全路径。 （相对于 `PGDATA` ） 替代记住许多上述规则，它往往是有用的。但是请记住， 这个函数只给了关系主分叉文件的第一部分的名称—你可能还需要一段 数字和/或 `_fsm` or `_vm` 用来找到关联该关系的所有文件。

创建临时文件（对于操作如更多数据于可适合内存的排序） 在 `PGDATA``/base/pgsql_tmp` ，或如果表空间不是指定的 `pg_default` ， 在表空间目录下的 `pgsql_tmp` 子目录。 临时文件的名表示为 `pgsql_tmp``_PPP_` . `_NNN_` ， 这里 `_PPP_` 是后台拥有的PID和 `_NNN_` 是后台来区分不同的临时文件。

58.2. TOAST

本节提供一个TOAST的概述。（超大字段存储技术）

因为PostgreSQL的页面大小是固定的(通常是8Kb)， 并且不允许行跨越多个页面，因此不可能直接存储非常大的字段值。 为了突破这个限制，大的字段值被压缩和/或打碎成多个物理行。 这些事情对用户都是透明的，只是在后端代码上有一些小的影响。 这个技术称为TOAST。（"切片面包之后最好的东西"))

只有一部分数据类型支持TOAST —(没必要在那些不可能生成大的字段值 的数据类型强制这种额外开销)。要支持TOAST，数据类型必须有变长 (*varlena*)表现形式，这个时候，任何存储的数值的头 32 位都是存储着以字节记的数值的总长度(包括长度本身)。 TOAST并不约束剩下的表现形式。所有支持TOAST的数据类型之 C级别的函数都必须仔细处理TOAST的输入值。也就是通常是在对一个输入值做任何事情之前调用 `PG_DETOAST_DATUM` ；但是在某些情况下也存在更高效的方法。

TOAST占用变长的长度字的两位（在大型机器上高位序，在小型机器上低位序）， 因此限制TOAST数据类型任何值的逻辑大小为1 GB（2³⁰ - 1字节）。当两位都是零时，该值是一个普通的非TOAST数据类型的值， 长度字的剩下位给总数据大小以字节计（包括长度字）。当设置最高或最低位， 该值仅有一个字节头替代通常的4字节头，而剩余的位给总数据大小以字节计（包括长度字）。 作为一个特殊的情况下，如果剩余位都是零（其将不可能包含自身的长度）， 该值为一个指向存储在TOAST表的行外数据。 （一个TOAST指针的大小是给定的在第二个字节的数据。） 单字节头的值没有对齐任何特定的边界。最后当清除最高或最低位时， 但是设置了临近的位，压缩了数据内容，在使用前必须解压缩。 在这种情况下长度字剩余位给压缩数据的总大小，而不是原数据的。 请注意压缩也可能是行外数据， 但是变长的头不会告诉这是否发生—反而TOAST指针的内容告诉这些。

如果一个表中有任何一个字段是可以TOAST的， 那么该表将有一个关联的TOAST表，其OID存储在表的 `pg_class` . `reltoastrelid` 记录里， 行外TOAST过的数值保存在TOAST表里，下面有更详细的描述。

这里使用的压缩技术是非常简单并且非常快速的 LZ 族压缩技术。 参阅 `src/backend/utils/adt/pg_lzcompress.c` 获取细节。

将外数据分割成(如果压缩过，在压缩之后)最多 `TOAST_MAX_CHUNK_SIZE` (缺省选择这个值，2000字节，使4块行将适合一内存页，约2000个字节)字节， 每个块都作为独立的行 在TOAST表里为所属表存储。 每个TOAST表都有 `chunk_id` 字段(一个表示特定TOAST值的OID)、 `chunk_seq` (一个序列号，存储该块在数值中的位置)、 `chunk_data` (该块实际的数据)。 在 `chunk_id` 和 `chunk_seq` 上有一个唯一索引，提供对数值的快速检索。 因此，一个表示行外TOAST值的指针数据需要存储要查阅的TOAST的OID 和特定数值的OID(它

的 `chunk_id`)。为了方便，指针数据还存储逻辑数据的尺寸 (原始的未压缩的数据长度)以及实际存储的尺寸 (如果使用了压缩，则两者不同)。 加上头部的长度字，一个TOAST指针数据的总尺寸是18字节， 不管它代表的数值的实际长度是多大。

TOAST代码只有在准备向某表中存储超过 `TOAST_TUPLE_THRESHOLD` 字节(通常是2KB)的行的时候才会触发。 TOAST代码将压缩和/或行外存储字段值， 直到数值比 `TOAST_TUPLE_TARGET` 字节（通常是2KB）短， 或者无法得到更好的结果的时候才停止。 在一个UPDATE操作过程中，未改变的字段值通常原样保存； 所以，如果UPDATE一个带有行外数据的行时，如果行外数据值没有变化， 那么将不会有TOAST开销存在。

TOAST代码识别四种不同的存储TOAST字段的策略：

- `PLAIN` 避免压缩或者行外的存储;此外，它禁止使用单字节的头变长类型。 这只是对那些不能TOAST的数据类型才有可能。
- `EXTENDED` 允许压缩和行外存储。 这是大多数TOAST的数据类型的缺省。首先将企图进行压缩， 如果行仍然太大，那么则进行行外存储。
- `EXTERNAL` 允许行外存储，但是不许压缩。 使用 `EXTERNAL` ，将令那些在 `text` 和 `bytea` 字段上的子字符串操作更快 (代价是增加了存储空间)，因为这些操作是经过优化的：如果行外数据没有压缩， 那么它们只会去抓取需要的部分。
- `MAIN` 允许压缩，但不允许行外存储。 实际上，在这样的字段上仍然会进行行外存储，但只是作为没有办法把数据行变得更小的情况下使之足以容纳一个页面的最后的手段。

每个TOAST的数据类型都为该数据类型的字段指定一个缺省策略， 但是特定表的字段的存储策略可以用 `ALTER TABLE SET STORAGE` 修改。

这个方法比那些更直接的方法，比如允许行值直接跨越多个页面， 有更多优点。假设查询通常是用相对比较短的键值进行匹配的， 那么大多数执行器的工作都将使用主行记录完成。 TOAST属性的大值， 只是在把结果集发送给客户端的时候才抽出来(如果选择了它的话)。因此， 主表要小得多，并且它的大部分行都存储在共享缓冲区里，因此就可以不需要任何行外存储。 排序集也缩小了，并且排序将更多地在内存里完成。一个小测试表明， 一个用于保存HTML页面以及它们的URL的表，包括TOAST表在内， 存储将近一半大小的裸数据，而主表只包含全部数据的10%(URL和一些小的HTML页面)。 与在一个非TOAST的对比表里面存储(把全部HTML页面裁剪成7KB以匹配页面大小)， 没有任何运行时的区别。

58.3. 自由空间映射

每个堆和索引关系，除了哈希索引， 有个自由空间映射(FSM) 来保持跟踪关系中可用的空间。 将同时在独立的关系叉文件存储主关系数据， 以关系的filenode 编号命名，加上一个 `_fsm` 后缀。 例如，如果一个关系的filenode是12345， 存储FSM在一个叫 `12345_fsm` 的文件里， 在与主关系文件相同目录里。

自由空间映射组织为一个FSM页树。 FSM页底层存储每个堆（或索引） 页上可用的自由空间， 使用一个字节来代表每一个如页。 高级别的从低级别聚合信息。

每个FSM页是一个二叉树，存储在一个数组， 每个节点一个字节。每个叶节点代表一个堆页，或低级别的FSM页。 在每个非叶节点，存储其子节点值的高级别的值。 因此在根节点存储叶节点的最大值。

参阅 `src/backend/storage/freespace/README` 关于更详细的FSM是怎样的结构， 怎样更新和搜索它。pg_freespacemap模块可以用来审查存储在自由空间映射的信息。

58.4. 可见映射

每个堆关系有个可见映射（VM）来保持跟踪那些包含行的页， 对于所有活动的事务可见。同时在独立的关系叉文件存储主关系数据， 以关系的filenode号，加上一个 `_vm` 后缀命名。例如， 如果一个关系的filenode是12345，存储VM在一个叫 `12345_vm` 文件里， 与主关系文件在同一目录。请注意索引没有VM。

可见映射在简单的在每个堆页存储1位。 一个设置位意味着在页上所有的行对于所有事务可见的。 这意味着不包含任何行的页，需要清理；使用*index-only scans* 回答仅仅使用索引元的查询也可以使用这些信息。

这个意义上的映射是保守的，我们要确定每当设置位， 我们知道条件是真，但是如果没有设置位，它可能是真，也可能不是真。 通过清理设置可见映射位，但是通过页上的任何数据修改操作进行清理。

58.5. 初始化分支

每个未记录的表，以及未记录表的每一个索引，有一个初始化 分支。初始化分支是一个空表或相应类型的索引。当一个未记录的表由于崩溃必须重置为空， 初始化分支被拷贝给主分支，并且擦除任何其他的分支（根据需要他们会自动重建）。

58.6. 数据库分页文件

本节提供一个在PostgreSQL表和索引使用的页格式的概述。[1] 序列和TOAST表的格式就像一个普通表的。

下面说明一下，一个字节假定为包含8位。另外，术语项为存储在页上的一个独立数据值。在表中，一项是一行；在索引中，一项为一个索引条目。

每个表和索引存储为固定大小的页数组。（通常 8 kB，不过当编译服务器的时候，可以选择不同的页大小） 在表中，所有的页是逻辑等价的，所以一个特殊项（行） 可以存储在任意页。在索引，第一页通常保留为持有控制信息的元页， 这里可以有不同类型的索引页，依赖于索引访问方法。

Table 58-2显示一个页的整体布局，这里每页有5部分。

Table 58-2. 页整体布局

项	描述
PageHeaderData	24字节长整型。包含关于页的一般信息，包含自由空间指针。
ItemIdData	指向实际项的（偏移量，长度）数组对。每项4字节。
Free space	未分配空间。从这个区域开始分配新项指针，或从结尾分配新项指针
Items	实际项本身
Special space	索引访问方法专用数据。不同方法存储不同的数据。普通表里为空。

每页的前24个字节构成一个页头（PageHeaderData）。 在Table 58-3有它的详细格式。 前两个字段跟踪相关页的最近的WAL条目。 下边的一个2字节的字段是包含标志位。 随后由3个2字节整数字段（ `pd_lower` ， `pd_upper` ，和 `pd_special` ）。 这些包含分别为从页开始到未分配空间的开始，到未分配空间的结束， 专用空间的开始的偏移字节数。下边页头的2字节，pd_pagesize_version， 存储页大小和版本指示符。 从PostgreSQL 8.3开始版本编号是4；PostgreSQL 8.1和8.2使用版本编号3； PostgreSQL 8.0使用版本编号2； PostgreSQL 7.3和7.4使用版本编号1； 先前发布版本使用版本编号0。（在大多数这些版本中，基本的页布局和头格式没有变化，但是堆布局有行头.） 页面大小是基本上只存在一个交叉检查；在安装的版本中， 这里不支持多于一页大小的。最后一个字段是个提示，显示是否整理页， 可能是有利的。它跟踪在页上最旧的未修整的XMAX。

Table 58-3. PageHeaderData布局

字段	类型	长度	描述
pd_lsn	XLogRecPtr	8字节	LSN: 该页上xlog日志记录变化的最后字节的下一字节
pd_tli	uint16	2字节	最后变化的时间线ID（仅其最低16位）
pd_flags	uint16	2字节	标志位
pd_lower	LocationIndex	2字节	到自由空间开始的偏移量
pd_upper	LocationIndex	2字节	到自由空间结尾的偏移量
pd_special	LocationIndex	2字节	到专用空间开始的偏移量
pd_pagesize_version	uint16	2字节	页大小和版本编号布局信息
pd_prune_xid	TransactionId	4字节	页上最旧的未修整的XMAX，如果没有则为零。

在 `src/include/storage/bufpage.h` 可以找到所有的详细信息。

下面的页头是项标识符（ `ItemIdData` ），每个需要4字节。 一个项标识符包含一个到项开始的字节偏移， 以字节计的长度，和一些影响它解释的属性位。 新项标识符需要从未分配空间的开始分配。 可以通过查看 `pd_lower` 来确定项标识符的数量，分配新的标示符， 其会增加。因为一个项标示符从来不移动直到释放了它，实际上， 每个指针为PostgreSQL所创建的一项由页号和项标识符的索引构成。 （ `ItemPointer` ，还可以称为 `CTID` ）。

项本身存储在从未分配的空间的结尾向后分配的空间。确切的结构取决于包含什么表。 表和序列两都使用一个名为 `HeapTupleHeaderData` 的结构，下面描述。

最后这段是"特殊段"其包含想存放的任何访问方法。 例如，b-tree索引存储连接页左右的兄弟，以及相应的索引结构的一些其它数据。 普通的表根本没有使用特殊段。 （通过设置 `pd_special` 等于页大小来表示）

所有表行结构方式相同。有个固定大小的头（在大多数机器占用23字节）， 随后一个NULL位图的可选项，对象ID字段，和用户数据。 该头的详细信息在Table 58-4。 实际的用户数据（行中列）由 `t_hoff` 表示的偏移量开始， 它必须始终是为平台的MAXALIGN间距的倍数。NULL位图仅存在，如果在 `t_infomask` 设置了*HEAP_HASNULL*位。 如果它存在，它就开始于固定头的后面，占用足够的字节，每数据列一位。 （那是， `t_natts` 位一块） 在这个位列表中，一个1位 标识非空，一个 0 位是空。 对象ID 仅存在，如果在 `t_infomask` 设置了*HEAP_HASOID*位。 如果存在，它将出现在t_hoff边界前。任何需要做 t_hoff 的MAXALIGN倍数的填充， 出现在NULL位图和对象ID之间。 （反过来又保证对象ID得到恰当的对齐）

Table 58-4. HeapTupleHeaderData布局

字段	类型	长度	描述
t_xmin	TransactionId	4字节	插入XID戳
t_xmax	TransactionId	4字节	删除XID戳
t_cid	CommandId	4字节	插入和/或 删除 CID戳(使用t_xvac覆盖)
t_xvac	TransactionId	4字节	VACUUM操作移动一行版本的XID
t_ctid	ItemPointerData	6字节	这个当前的或新行版本的TID
t_infomask2	uint16	2字节	字段个数，加上各种标志位
t_infomask	uint16	2字节	各种标志位数
t_hoff	uint8	1字节	用户数据偏移量

在 `src/include/access/htup.h` 可以找到所有的详细信息。

解释实际数据只能从其它表获取信息来做，大多 `pg_attribute` 。 需要来表示字段位置的键值是 `attlen` 和 `attalign` 。 没有直接获取特定字段的方法，除仅当有固定宽度字段并且没有空值的情况外。 所有这些策略封装在函数*heap_getattr*，*fastgetattr* 和*heap_getsysattr*。

要读取数据你需要逐次检查每个属性。首先检查字段是否为NULL依据NULL位图。 如果是，跳到下一个。然后确定你已经右对齐。如果字段是固定宽度的字段， 那么所有的字节简单的放置。如果它是变长的字段（attlen = -1） 那么它是一个更复杂的位。所有变长数据类型共享通用的头结构 `struct varlena` ， 其包括存储值的总长度和一些标志位。 依赖这些标志，数据可能是行内或在一个TOAST表；它也可能是压缩的。 （参阅Section 58.2）

Notes

[1] 实际上，索引访问方法不需要使用这个页格式。所有已经存在的索引方法 需要使用基本格式， 但是保持在索引元页上的数据通常不遵循项布局规则。

Chapter 59. BKI后端接口

Table of Contents

后端接口(BKI)文件是一些用特殊语言写的脚本，这些脚本是 PostgreSQL后端能够理解的，以特殊的 "bootstrap" (引导)模式运行，这种模式允许在不存在系统表的零初始条件下执行数据库函数， 而普通的 SQL 命令要求系统表必须存在。因此BKI 文件可以用于在第一时间创建数据库系统。（可能除此以外也没有其它用处。）

在创建一个新的数据库集群的时候，initdb就是使用BKI 文件来完成其工作的一部分。initdb使用的输入的文件是作为编译和安装 PostgreSQL的一部分，由一个叫 `genbki.pl` 的程序创建的，这个程序读取源代码树目录的 `src/include/catalog/` 目录里面的几个特殊格式的 C 头文件。生成的BKI文件叫 `postgres.bki` 并且通常安装在安装树里的 `share` 子目录。

相关的信息可以在有关initdb的文档中找到。

59.1. BKI 文件格式

本节描述PostgreSQL后端是如何理解BKI文件。 如果把 `postgres.bki` 文件拿来做为例子，这些描述会变得容易理解些。

BKI输入是由一系列命令组成的。命令是由一些记号组成的， 具体是什么由命令语法决定。记号通常是用空白分隔的，但是如果没有歧义的话可以不要。 没有什么特殊的命令分隔符；语法上无法属于前面命令的记号开始新的一条命令。 （通常你会把一条新的命令放在新的一行上以保持清晰。）记号可以是某些关键字， 特殊字符(圆括弧，逗号等)，数字，或者双引号字符串。所有东西都是大小写敏感的。

以 `#` 开头的行被忽略。

59.2. BKI 命令

`create` `_tablename_` `_tableoid_` [`bootstrap`] [`shared_relation`] [`without_oids`] [`rowtype_oid` `_oid_`] (`_name1_` = `_type1_` [, `_name2_` = `_type2_` , ...])

创建一个名为 `_tablename_` 并且 OID 为 `_tableoid_` 的表，表字段在圆括弧中给出。

`bootstrap.c` 直接支持下列字段类型： `bool` , `bytea` , `char` (1 字节), `name` , `int2` , `int4` , `regproc` , `regclass` , `regtype` , `text` , `oid` , `tid` , `xid` , `cid` , `int2vector` , `oidvector` , `_int4` (数组), `_text` (数组), `_oid` (数组), `_char` (数组), `_aclitem` (数组)。 尽管可以创建包含其它类型字段的表，但是只有在创建完 `pg_type` 并且填充了合适的记录之后才行。 这实际上就意味着在系统初始化表中，只能使用这些字段类型，而非系统初始化表可以使用任意内置类型。

如果声明了 `bootstrap` ，那么将只在磁盘上创建表；不会向 `pg_class` , `pg_attribute` 等系统表里面输入任何东西。因此这样的表将无法被普通的 SQL 操作访问， 直到那些记录用硬办法(用 `insert` 命令)填入。这个选项用于创建 `pg_class` 等自身。

如果声明了 `shared_relation` ， 那么表就作为共享表创建。除非声明了 `without_oids` ， 否则将会有 OID 。表的行类型OID（(`pg_type` OID）可以可选的通过 `rowtype_oid` 子句声明；如果没有声明，会自动为表生成一个OID。（如果声明了 `bootstrap` ， 那么 `rowtype_oid` 子句是没有用的，但是它可以在文档中给出。）

`open` `_tablename_`

打开一个名为 `_tablename_` 的表，准备插入数据。 任何当前已经打开的表都会被关闭。

`close` [`_tablename_`]

关闭打开的表。给出的表名用于交叉检查，但并不是必须的。

`insert` [`OID =` `_oid_value_`] `(` `_value1_` `_value2_` ... `)`

如果 `_oid_value_` 为零，那么用 `_value1_` , `_value2_` 等作为字段值以及 `_oid_value_` 作为其OID(对象标识)向打开的表插入一条新记录， 否则省略子句，而表则拥有 OID ，并赋予下一个可用的 OID 数值。

NULL 可以用特殊的关键字 `_null_` 声明。包含空白的值必须用双引号括起。

`declare` [`unique`] `index` `_indexname_` `_indexoid_` `on` `_tablename_` `using` `_amname_` `(` `_opclass1_` `_name1_` [, ...] `)`

在一个叫 `_tablename_` 的表上用 `_amname_` 访问方法创建一个 OID 是 `_indexoid_` 的叫做 `_indexname_` 的索引。索引的字段叫 `_name1_` , `_name2_` 等，而使用的操作符类分别是 `_opclass1_` , `_opclass2_` 等。将会创建索引文件和恰当的系统表记录， 但是索引内容不会

被此命令初始化。

`declare toast` `_toasttableoid_` `_toastindexoid_` `on` `_tablename_`

为名为`_tablename_`的表创建一个 TOAST 表。 这个 TOAST 的 OID 是`_toasttableoid_`，其索引的 OID 是`_toastindexoid_`。 与`declare index`一样，索引的填充会被推迟。

`build indices`

填充前面声明的索引。

59.3. 系统初始化的BKI文件的结构

`open` 命令打开的表需要系统事先存在另外一些基本的表，在这些表存在并拥有数据之前，不能使用 open 命令。这些最低限度必须存在的表是 `pg_class`，`pg_attribute`，`pg_proc`，`pg_type`。为了允许这些表自己被填充，带 `bootstrap` 选项的 `create` 隐含打开所创建的表用于插入数据。

同样，`declare index` 和 `declare toast` 命令也不能在它们所需要系统表创建并填充之前使用。

因此，`postgres.bki` 文件的结构必须是这样的：

1. `create bootstrap` 其中一个关键表
2. `insert` 数据，这些数据至少描述这些关键表本身
3. `close`
4. 重复创建和填充其它关键表
5. `create` (不带 `bootstrap`)一个非关键表
6. `open`
7. `insert` 需要的数据
8. `close`
9. 重复创建其它非关键表
10. 定义索引
11. `build indices`

当然，肯定还有其它未记录文档的顺序依赖关系。

59.4. 例子

下面的命令集将创建OID为420名为 `test_table` 的表， 该表有两个类型分别为 `int4` 和 `text` 的字段 `cola` 和 `colb` ， 然后向该表插入两行。

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Chapter 60. 规划器如何使用统计信息

本章建立在 Section 14.1和Section 14.2 里面讨论的材料上， 显示了关于规划器如何使用的额外的详细信息。 系统统计信息来预计一个查询运行的各个阶段可能返回的行数。 这是规划过程中的一个重要的部分， 因为它提供了开销计算中的大部分原始材料。

本章的目的不是在细节上给代码写文档(代码本身就是文档)， 而是给出一个规划器如何使用统计信息的概述。 这样可能可以降低那些想以后阅读这部份代码的人的学习难度。

60.1. 行预期的例子

下面的例子使用的PostgreSQL回归测试数据库中的表。 输出结果是从8.3版获得的。之前或之后版本的动作可能会有所变化。 同时需要注意的是，在产生统计信息时， `ANALYZE` 使用的是随机采样， 在使用一次新的 `ANALYZE` 之后，结果可能会发生轻微的改变。

让我们以一个很简单的查询开始：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

规划器如何判断 `tenk1` 里面行的基数 在Section 14.2里面介绍， 为了完整，在这里重复一下。行数或页数是从 `pg_class` 里面查出来的：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

这些数字表示表中当前最新的 `VACUUM` 或者 `ANALYZE` 。 之后，规划器取出表中当前实际的块号（这个操作的开销很小，不需要扫描全表） 。 如果与 `relpages` 不同，那么根据达到的一个当前函数估计值， `reltuples` 会进行一定的缩放。 在这种情况下， `relpages` 的值是最新的，因此估计的行与 `reltuples` 相同。

换一个在WHERE子句里面带有范围条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器检查 `WHERE` 子句条件， 并为 `pg_operator` 中的 < 执行器查找可选函数。 将保持在 `oprrest` 列中， 并且在这个例子中的条目是 `scalarltsel` 。 `scalarltsel` 函数从 `unique1` 为 `unique1` 检索直方图。 对于手工查询来说，这样做可以更方便，更直观的查看 `pg_stats` 视图：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                   histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

然后，把直方图里面包含"< 1000"的部分找出来。这就是选择性。 直方图把范围分隔成相同频率的段， 所以要做的只是把的数值所在的段找出来， 然后计算它里面占的部分以及所有该值之前的部分。 值1000很明显在第二个段(993-1997)里，因此，假设每个段里面的分布是线性的， 那么就可以计算出选择性：

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

也就是一个段加上第二个段的线性部分，除以总段数。 那么估计的行数现在可以用选择性和 `tenk1` 的基数之积计算：

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

然后考虑一个 `WHERE` 子句里等于条件的例子：

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

规划器再次检查 `WHERE` 子句条件，并为 `=` （是 `eqsel` ）查找可选函数。 对于等价估计而言，直方图并不是有用的；相反， 最常见的值(MCV)列表可以用来决定可选项。 让我们来看一下MCV，带有一些额外的列会很有效：

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAA
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

因为MCV中有 `CRAAAA` ，那么可选项只是MCF列表中的一个相关条目：

```
selectivity = mcf[3]
            = 0.003
```

像之前一样，行的估计数只是和前面一样用 `tenk1` 的基数乘以选择性：

```
rows = 10000 * 0.003
     = 30
```

现在看看同样的查询，但是字符串常量是不在MCV列表里的:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

这个时候的问题是完全不同的一个：在数据值不在 MCV列表里面时， 如何估计选择性就是完全另外一个问题了。解决方法是利用该值不在列表里头的事实， 结合已知的所有MCV出现的频率，用减法得出：

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

也就是，为MCV增加所有的频率，并且从1减去， 然后用其它无重复值的个数来分开。 这相当于假设不是MCV中的列的分数巨晕的分布在所有其他不同值中。 需要注意的是，没有NULL值，因此不需要担心这些（否则需要从分子中减去NULL分数）。 估算的行数然后照例计算：

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

之前带有 `unique1 < 1000` 的例子是 `scalarltsel` 实际执行的简单化。 现在已经看过了使用MCV的例子，可以增加一些具体细节了。 这个例子这样子是正确的，因为 `unique1` 是一个唯一属性列， 那么它没有MCV（显然，没有一个值能比其它值更通用）。 对一个非唯一属性列而言，通常会有直方图和MCV列表， 并且直方图不包括MCV表示的列总体那部分。在这种情况下， `scalarltsel` 直接应用条件到每个 MCV列表的值上（如"< 1000"）， 并且增加那些条件判断为真的MCV的频率。这给出准确的是MCV表的部分的选择的准确估计。 然后直方图使用与上述方式相同的估计选择表的部分，其不是MCV， 那么组合这两个数字来估计总的选择性。例如，考虑

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

我们已看到关于stringu1的MCV信息，这里是它的直方图：

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
--------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

检查MCV列表，我们发现前6项满足条件 `stringu1 <'IAAAAA'` ，而不是最后4项， 所以最常见的部分MCV选择性是

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

累加所有的MCF，也告诉我们由 MCVs表示的常见的总比例是0.03033333， 而且因此由直方图表示的 比例是0.96966667。 （再次,没有NULL，否则这里我们排斥它们）我们可以看到 `IAAAAA` 值落在第三段直方图的结尾部分。 关于不同字符串的频率使用些较普通的假设，规划器达到估计0.298387为 直方图中小于 `IAAAAA` 的部分。我们然后组合估计值为MCV和非MCV常见:

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

尤其是在这个例子中，MCV列表的纠正很小，因为列分布实际上很平坦。 （统计分析显示这些特殊值往往比其它的更常见大部分由于抽样误差） 在更典型的情况下这里有些值显著的比其它的更常见，这复杂的处理过程， 有用的提高了精度，因为选择性对于那些最常见的值来说，查找准确。

现在考虑一个 `WHERE` 子句中带有多个条件的情况：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

规划器认为这两个条件是独立的，因此可以同时执行语句的独立查询：

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

需要注意的是，从位图索引扫描中返回的估计行数值影响索引使用的条件； 这一点很重要，因为它会影响之后的堆栈抓取估计开销。

最后检查一个包含连接的查找：

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在 `tenk1` 上的 `unique1 < 50` 限制在嵌套循环连接之前计算。 这个条件是用类似上面的那个范围例子的方法处理的。 但是这次数值50落在 `unique1` 的直方图表的第一个段内：

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

此链接的限制是 `t2.unique2 = t1.unique2` 。操作符是我们熟悉的 `=` ， 然而可选函数是从 `pg_operator` 的 `oprjoin` 字段获得的， 并且是 `eqjoinsel` 。 `eqjoinsel` 为 `tenk2` 和 `tenk1` 查找统计信息：

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

在这个例子里，没有 `unique2` 的MCV信息， 因为所有数值看上去都是唯一的， 因此可以使用一个只依赖唯一数值数目和NULL数目百分比的算法来给两个表计算(选择性)：

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

也就是说，把每个表都减去一里面NULL的比例，然后除以数值的最大值。 连接可能选出来的行数是以嵌套循环里的两个输入值的笛卡尔积的总行数， 乘以选择性计算出来的：

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

这里有两列的MCV列表， `eqjoinsel` 将直接使用MCV列表比较来决定连接由MCV表示的常见列部分的选择。 下面常见的剩下的估计值跟显示这里的方法相同。

需要注意的是， `inner_cardinality` 表示为10000，也就是未修改的 `tenk2` 大小。 它可能出现从检查 `EXPLAIN` 输出，其连接行的估计值来自50 * 1，就是， 由外部行数乘以由每个内部索引扫描的 `tenk2` 获取的估计行数。但是这不是那种情况： 估计连接关系的大小在考虑任何特定的连接计划之前。如果任何事情工作很好， 那么两种方式估计的连接大小将产生相关的同样的答案， 但是由于四舍五入误差和其它因素它们有时差异较明显。

在 `src/backend/optimizer/util/plancat.c` 中有对一个表大小的估计 （在任何 `WHERE` 字句之前）。 在 `src/backend/optimizer/path/clausesel.c` 中有对字句选择性的通用逻辑。 在 `src/backend/utils/adt/selfuncs.c` 中有特定操作符的可选函数。

VIII. 附录

Table of Contents

Appendix A. PostgreSQL 错误代码

PostgreSQL服务器发出的所有消息都赋予了五个字符的错误代码， 这些代码遵循 SQL 的"SQLSTATE"代码的习惯。 需要知道发生了什么错误条件的应用通常应该测试错误代码，而不是查看文本错误信息。 这些错误代码轻易不会随着PostgreSQL的版本更新而修改， 并且一般也不会随着错误信息的本地化而发生修改。请注意有些(但不是全部) PostgreSQL生成的错误代码是由 SQL 标准定义的； 有些标准没有定义的错误条件是发明的或者是从其它数据库借来的。

根据标准，错误代码的头两个字符表示错误类别，而后三个字符表示在该类别内特定的条件。 因此，那些不能识别特定错误代码的应用仍然可以从错误类别中推断要做什么。

Table A-1里面列出了PostgreSQL 9.3.1 定义的所有错误代码(有些实际上目前并没有使用，但是 SQL 标准定义了)。错误类别也列出在此。 对于每个错误类别都有个"标准"的错误代码，它的最后三个字符是 `000` 。 这个代码只用于那些落在该类别内，但是没有赋予任何更准确的代码的错误条件。

"条件名"列显示的标志是在PL/pgSQL里面使用的条件名。 条件名大小写无关。 （请注意PL/pgSQL并不识别警告， 这一点和错误、条件名正相反；那些是 00，01，02 类别。）

对于某些类型的错误，服务器报告与错误相关的数据库对象的名称（一个表，表字段，数据类型， 或者常量）；例如，唯一约束的名字导致了一个 `unique_violation` 错误。 类似的名字在单独的错误报告字段信息中提供， 这样应用程序就不用尝试去从可能的本地化人可读的信息文本中提取它们。 自PostgreSQL 9.3起，对这个信息的完全覆盖只存在于SQLSTATE类23的错误中（完整性约束违反）， 但这个可能在将来扩展。

Table A-1. PostgreSQL 错误代码

错误代码	条件名
Class 00 — Successful Completion	
`00000`	`successful_completion`
Class 01 — Warning	
`01000`	`warning`
`0100C`	`dynamic_result_sets_returned`
`01008`	`implicit_zero_bit_padding`
`01003`	`null_value_eliminated_in_set_function`
`01007`	`privilege_not_granted`
`01006`	`privilege_not_revoked`
`01004`	`string_data_right_truncation`

01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	

0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch

22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction

25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
Class 26 — Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted

38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name

42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation

Class 53 — Insufficient Resources	
`53000`	`insufficient_resources`
`53100`	`disk_full`
`53200`	`out_of_memory`
`53300`	`too_many_connections`
`53400`	`configuration_limit_exceeded`
Class 54 — Program Limit Exceeded	
`54000`	`program_limit_exceeded`
`54001`	`statement_too_complex`
`54011`	`too_many_columns`
`54023`	`too_many_arguments`
Class 55 — Object Not In Prerequisite State	
`55000`	`object_not_in_prerequisite_state`
`55006`	`object_in_use`
`55P02`	`cant_change_runtime_param`
`55P03`	`lock_not_available`
Class 57 — Operator Intervention	
`57000`	`operator_intervention`
`57014`	`query_canceled`
`57P01`	`admin_shutdown`
`57P02`	`crash_shutdown`
`57P03`	`cannot_connect_now`
`57P04`	`database_dropped`
Class 58 — System Error (errors external to PostgreSQL itself)	
`58000`	`system_error`
`58030`	`io_error`
`58P01`	`undefined_file`
`58P02`	`duplicate_file`
Class F0 — Configuration File Error	
`F0000`	`config_file_error`
`F0001`	`lock_file_exists`

Class HV — Foreign Data Wrapper Error (SQL/MED)	
`HV000`	`fdw_error`
`HV005`	`fdw_column_name_not_found`
`HV002`	`fdw_dynamic_parameter_value_needed`
`HV010`	`fdw_function_sequence_error`
`HV021`	`fdw_inconsistent_descriptor_information`
`HV024`	`fdw_invalid_attribute_value`
`HV007`	`fdw_invalid_column_name`
`HV008`	`fdw_invalid_column_number`
`HV004`	`fdw_invalid_data_type`
`HV006`	`fdw_invalid_data_type_descriptors`
`HV091`	`fdw_invalid_descriptor_field_identifier`
`HV00B`	`fdw_invalid_handle`
`HV00C`	`fdw_invalid_option_index`
`HV00D`	`fdw_invalid_option_name`
`HV090`	`fdw_invalid_string_length_or_buffer_length`
`HV00A`	`fdw_invalid_string_format`
`HV009`	`fdw_invalid_use_of_null_pointer`
`HV014`	`fdw_too_many_handles`
`HV001`	`fdw_out_of_memory`
`HV00P`	`fdw_no_schemas`
`HV00J`	`fdw_option_name_not_found`
`HV00K`	`fdw_reply_handle`
`HV00Q`	`fdw_schema_not_found`
`HV00R`	`fdw_table_not_found`
`HV00L`	`fdw_unable_to_create_execution`
`HV00M`	`fdw_unable_to_create_reply`
`HV00N`	`fdw_unable_to_establish_connection`
Class P0 — PL/pgSQL Error	
`P0000`	`plpgsql_error`
`P0001`	`raise_exception`
`P0002`	`no_data_found`
`P0003`	`too_many_rows`
Class XX — Internal Error	
`XX000`	`internal_error`
`XX001`	`data_corrupted`

XX002	index_corrupted

Appendix B. 日期/时间支持

Table of Contents

PostgreSQL使用一个内部的启发式分析器用于所有的日期/时间支持。 日期和时间都是以字符串形式输入的， 然后用一个初步的判断分解为在该数域里可以有什么样的信息。 每个数域都被解释，并且要么是被赋予一个数字值，要么是忽略， 要么是被拒绝。分析器里包含内部的查询表，用于所有文本域， 包括月份、 星期几、时区。

这份附录包含这些查询表的信息，以及描述了分析器用来对时间和日期解码的步骤。

B.1. 日期/时间输入解析

日期/时间类型输入都是使用下列过程进行解码的。

1. 把输入的字符串分解为一个个记号，然后把每个记号分成字符串、 时间、时区、数字几类：

   i. 如果一个数字记号包含一个冒号(`:`)， 那么这是一个时间字符串。包括随后所有的数据位和冒号。

   ii. 如果这个数字记号包含一个划线(`-`)、斜杠(`/`)、 多个点(`.`)， 那么它就是一个日期字符串，可能有一个文本月份。 如果一个日期记号已经看过，那么将被解析为时区名(比如 `America/New_York`)。

   iii. 如果这个记号只是数字，那么它要么是一个单独的字段， 要么是一个ISO8601连接的日期(比如 `19990113` 是1999年1月13日) 或者是连接的时间(比如 `141516` 是14:15:16)。

   iv. 如果记号以一个加号(`+`)或减号(`-`)开头， 那么它要么是一个时区，要么就是一个特殊的字段。

2. 如果记号是一个文本字符串，那么和可能的字符串进行匹配：

   i. 把这个记号当作时区缩写进行二分表查找。

   ii. 如果没有找到，再做一次二分表查找，看看这个记号是特殊字符串 (比如 `today`)、日期(比如 `Thursday`)、 月份(比如 `January`)， 还是一个无关痛痒的字 (比如 `at` , `on`)。

   iii. 如果还没有找到，抛出一个错误。

3. 如果记号是一个数字或者数字字段：

   i. 如果有八位或者六位数字，而且前面也没有读到其它日期字段， 那么就解释成一个"concatenated date(连接的日期)" (比如 `19990118` 或者 `990118`)。 这里的解析是 `YYYYMMDD` 或者 `YYMMDD` 。

   ii. 如果记号是三位数字，并且已经解码了一个年份，那么解释成一年中的日。

   iii. 如果已经读取了四位或六位数字， 并且已经读取了一个年份， 那么就解析成时间(`HHMM` 或者 `HHMMSS`).

   iv. 如果是三位或更多位并且还没有找到日期字段，则解析成一个年份 (这个解析强制剩余的日期字段的顺序为yy-mm-dd)。

v. 否则，日期字段的顺序被认为是遵循 `DateStyle` 设置： mm-dd-yy,dd-mm-yy,yy-mm-dd 之一。如果发现月份或者日期字段超出范围， 则抛出一个错误。

4. 如果声明了BC，则对年份取其负数并加一，用于内部保存。 因为在格里高利历法里没有零年，所以数字上的1BC是公元零年。

5. 如果没有声明BC并且年份字段有两个数据位的长度，那么把年份调整为4位。 如果该字段小于70，那么加2000，否则加1900。

   > **Tip:** 格里高利年份AD 1-99可以用前导零的方式使用4位数字 (也就是说 `0099` 是AD 99)。

B.2. 日期/时间关键字

Table B-1显示被当做月份名字缩写的记号。

Table B-1. Month Names

月份	缩写
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

Table B-2显示被识别为星期几的名字。

Table B-2. Day of the Week Names

星期	缩写
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table B-3显示用于各种修饰用途的记号。

Table B-3. 日期/时间字段修饰词

标识符	描述
`AM`	12:00之前的时间
`AT`	忽略
`JULIAN` , `JD` , `J`	下一个字段是儒略日
`ON`	忽略
`PM`	12:00以及之后的时间
`T`	下一个字段是时间

B.3. 日期/时间配置文件

一些时区缩写并不很标准，PostgreSQL提供 一种定制服务器可接受的缩写集合。 timezone_abbreviations 运行时配置参数定义缩写的有效集合。 该配置参数可以被任何数据库用户更改， 但是其取值范围只能由数据库管理员更改， 事实上可用的值都是 `.../share/timezonesets/` 目录中的文件名。 通过添加或修改其中的文件，管理员就可以控制可用的时区缩写。

`timezone_abbreviations` 可以被设为 `.../share/timezonesets/` 目录下的任意文件名(文件名只允许包含字母)。 禁止在 `timezone_abbreviations` 中使用非字母字符是为了防止读取目录之外的文件以及其它不该读取的文件。

时区缩写文件中可以包含空白行和以 `#` 开头的注释。 非注释行必须是以下格式：

```
_time_zone_name_ _offset_
_time_zone_name_ _offset_ D
@INCLUDE _file_name_
@OVERRIDE
```

`_time_zone_name_` 是被定义的缩写名。 `_offset_` 是该时区相对于 UTC 偏移量(以秒计)， 向东为正，向西为负。例如-18000表示在格林威治以西5小时， 也就是美国东部标准时间。 `D` 表示该时区使用夏令时而不是标准时。 因为目前所有已知的时区偏移量都以15分钟为单位， 因此偏移量的秒数必须是900的倍数。

`@INCLUDE` 语法用于包含 `.../share/timezonesets/` 目录中的其它文件，可以嵌套包含，不过并不允许无限深度的嵌套。

`@OVERRIDE` 语法表示后面项的定义可以覆盖前面的项（比如，从包含文件中获得）， 没有这一点，同一时区缩写的矛盾定义被认为是一个错误。

默认安装时， `Default` 文件包含世界上几乎所有不冲突的时区缩写。 另外， `Australia` 和 `India` 文件用于这些区域： 这些文件首先被包含在 `Default` 文件中并在随后按需修改或者添加时区。

为了便于参考，标准安装也包含了 `Africa.txt` ， `America.txt` 等文件， 它们包含了所有 `zoneinfo` 时区数据库中的时区缩写。 这些文件中的时区名定义可以复制到自定义的配置文件中。 需要注意的是，这些文件名不能直接用于 `timezone_abbreviations` ， 因为这些文件名中包含句点。

> **Note:** 如果在读取时区数据集时出错， 将不会应用任何新值，仍将使用旧的数据集。 如果这个错误是在数据库服务器启动时发生的，那么启动将失败。

Caution

配置文件中的时区缩写定义将会覆盖PostgreSQL内置的非时区含义。 例如`Australia`配置文件定义了`SAT`(南澳洲标准时间)， 如果激活了该文件，那么`SAT`将不会被识别为星期六的缩写。

Caution

如果你修改`.../share/timezonesets/`中的文件， 那么你必须自己手动备份，因为数据库转储不会包含这个目录的内容。

B.4. 单位历史

该SQL标准指出"在一个'日期时间形式'定义中， '日期时间值'按照阳历 通过日期和时间的自然规则被限制"。 PostgreSQL遵循SQL标准通过阳历计算特定日期，甚至使用日历的几年前。这条规则被称为预期的阳历。

儒略日是由Julius Caesar在公元前45年引入的，直到1582年开始使用公历之前， 西方国家一直使用儒略日。 在儒略日中，一年近似等于365+1/4=365.25天， 大约在128年的出现一个1天的错。

不断积累的历法错误促使教皇格里高利十三世(Gregory XIII)按照与 弥撒议会(Council of Trent)一致的精神改革了历法。在罗马历法里， 一年是近似365 + 97 / 400天= 365.2425天。因此对应于罗马历法，大约要3300年，才会积累一天的误差。

近似的365+97/400是通过利用下面的规则，规定每400年有97个闰年实现的：

每个可被**4**整除的年是一个闰年
不过，可被100整除的年不是闰年
但是，可以被400整除的年还是闰年。

因此，1700,1800,1900,2100和2200年都不是闰年。而1600,2000,2400年是闰年。 相比而言，旧式的Julian历法里面只有能被4整除的年是闰年。

罗马教皇在1582年2月宣布从1582的10月中删除10天， 也就是10月15号紧跟在10月4号的后面。 信奉天主教的国家(意大利、波兰、葡萄牙、西班牙等) 很快就遵循了这个规定，但新教国家拒绝使用， 而希腊东正教国家却一直拖延到20世纪开始的时候才逐渐遵守这个规定。 大英帝国及其殖民地(包含今天的美国)在1752年开始引用使用， 也就是1752年9月2号之后紧跟着14号， 这就是为什么Unix系统上的 `cal` 程序会产生如下输出的原因：

```
$ <kbd class="literal">cal 9 1752</kbd>
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

但是，当然，这个日历对于英国领土是唯一有效的，不是其它地方。 因为尝试跟踪在不同的时间不同地方使用的实际日历是困难和混乱的， PostgreSQL不会尝试，而是遵循所有日期的阳历 规则，即使这种方法是历史上不准确的。

在世界的不同地方，发明了许多不同的历法，有许多比罗马历法系统还早。 例如，中国历法的最早应用可以追溯到公元前14世纪。 传说黄帝在公元前2637就发明了这个历法，也就是日历。 中华人民共和国使用罗马历法用于民用。 中国历法用于决定节日/节气。

*Julian Date*系统是日历的另一种类型， 跟Julian calendar无关，尽管与这个日历类似命名是令人费解的。 "Julian Date"系统是法国学者Joseph Justus Scaliger(1540-1609)发明的， 可能是取自Scaliger的父亲的名字， 意大利学者Julius Caesar Scaliger(1484-1558)。在Julian日期系统， 每天产生一个序列号，从JD0开始（有时被叫做the Julian日期）。 JD0在Julian日历中对应公元前4713年1月1日，在Gregorian日历中对应公元前4714年11月24日。 Julian日期计算经常在天文学家标注夜间观测时被用到， 因此一个日期就是从一个正午UTC到下一个正午UTC。 而不是从午夜到另一个午夜： JD0设计的24小时是从公元前4714年11月24日的正午UTC到公元前4714年11月25日的正午UTC。

尽管PostgreSQL在输入输出日期时支持Julian Date日期符号 （也用在一些内部的日期时间日历上）， 它不观察从正午到正午的精密运行。 PostgreSQL运行Julian Date日期系统从午夜到午夜。

Appendix C. SQL关键字

Table C-1列出了所由在 SQL 标准和PostgreSQL 里是关键字的记号。你可以在Section 4.1.1里找到相关的背景信息。 （由于空间原因，只包括SQL标准的最后两个版本和为了历史兼容的SQL-92。 这些和其他中间标准版本的差异很小。）

SQL 里有保留字和非保留字之分。根据标准， 保留字是那些真正的关键字；决不能用它们做标识符。非保留字只是在特定的环境里有特殊的含义， 而在其它环境里是可以用做标识符的。大多数非保留字实际上都是 SQL 声明的内建表和函数的名字。 而非保留字的概念实质上只是用来表示在某些场合里，在一个字上附加了一些预先定义的含义。

在PostgreSQL里，分析器的工作有些复杂。因为存在好几种不同范畴的记号， 从那些永远不可能用做标识符的到那些和普通标识符比较起来在分析器里完全没有任何特殊状态的 (后者通常都是 SQL 声明的函数)。即使保留关键字在PostgreSQL 里都没有完全保留，而是可以用做字段标签。比如，虽然 `CHECK` 是保留关键字， 但是 `SELECT 55 AS CHECK` 语句完全行得通。

在Table C-1的PostgreSQL字段里， 我们把那些分析器明确知道，但是允许作为字段或表名的关键字分类为"非保留"。 虽然一些关键字是非保留的，但是却不能用做函数或者数据类型名字，我们对这些关键字相应做了的标记。 大多数这类记号代表有特殊语法的内置函数或者数据类型。仍然可以使用这些函数或者类型， 但是用户不能重新定义它们。标记为"保留"的都是那些不允许作为字段或表名的。 有些保留关键字还可以用做函数或数据类型的名字；这点在表中也显示出来了。 如果没有那样标记，那么只允许出现在"AS"字段标签名里面。

一条通用的规则是：如果你看到任何古怪的分析错，说命令包含任何这里列出的关键字做标识符， 那么你可以先试试用双引号把那个标识符括起来，看看问题是否消失。

在开始学习Table C-1之前还要理解的一件重要的事情是： 一个关键字在PostgreSQL 中没有保留并不意味着与该关键字相关的特性没有实现。同样， 关键字的存在也并不表明某个特性就一定存在。

Table C-1. SQL 关键字

关键字	PostgreSQL	SQL:2011	SQL:2008	SQL-92
`A`	非保留	非保留		
`ABORT`	非保留			
`ABS`	保留	保留		
`ABSENT`	非保留	非保留		
`ABSOLUTE`	非保留	非保留	非保留	保留
`ACCESS`	非保留			

ACCORDING	非保留	非保留		
ACTION	非保留	非保留	非保留	保留
ADA	非保留	非保留	非保留	
ADD	非保留	非保留	非保留	保留
ADMIN	非保留	非保留	非保留	
AFTER	非保留	非保留	非保留	
AGGREGATE	非保留			
ALL	保留	保留	保留	保留
ALLOCATE	保留	保留	保留	
ALSO	非保留			
ALTER	非保留	保留	保留	保留
ALWAYS	非保留	非保留	非保留	
ANALYSE	保留			
ANALYZE	保留			
AND	保留	保留	保留	保留
ANY	保留	保留	保留	保留
ARE	保留	保留	保留	
ARRAY	保留	保留	保留	
ARRAY_AGG	保留	保留		
ARRAY_MAX_CARDINALITY	保留			
AS	保留	保留	保留	保留
ASC	保留	非保留	非保留	保留
ASENSITIVE	保留	保留		
ASSERTION	非保留	非保留	非保留	保留
ASSIGNMENT	非保留	非保留	非保留	
ASYMMETRIC	保留	保留	保留	
AT	非保留	保留	保留	保留
ATOMIC	保留	保留		
ATTRIBUTE	非保留	非保留	非保留	
ATTRIBUTES	保留	保留		
AUTHORIZATION	保留（可以是函数或类	保留	保留	保留

	型）			
AVG	保留	保留	保留	
BACKWARD	非保留			
BASE64	非保留	非保留		
BEFORE	非保留	非保留	非保留	
BEGIN	非保留	保留	保留	保留
BEGIN_FRAME	保留			
BEGIN_PARTITION	保留			
BERNOULLI	非保留	非保留		
BETWEEN	非保留（不能是函数或类型）	保留	保留	保留
BIGINT	非保留（不能是函数或类型）	保留	保留	
BINARY	保留（可以是函数或类型）	保留	保留	
BIT	非保留（不能是函数或类型）	保留		
BIT_LENGTH	保留			
BLOB	保留	保留		
BLOCKED	非保留	非保留		
BOM	非保留	非保留		
BOOLEAN	非保留（不能是函数或类型）	保留	保留	
BOTH	保留	保留	保留	保留
BREADTH	非保留	非保留		
BY	非保留	保留	保留	保留
C	非保留	非保留	非保留	
CACHE	非保留			
CALL	保留	保留		
CALLED	非保留	保留	保留	
CARDINALITY	保留	保留		

CASCADE	非保留	非保留	非保留	保留
CASCADED	非保留	保留	保留	保留
CASE	保留	保留	保留	保留
CAST	保留	保留	保留	保留
CATALOG	非保留	非保留	非保留	保留
CATALOG_NAME	非保留	非保留	非保留	
CEIL	保留	保留		
CEILING	保留	保留		
CHAIN	非保留	非保留	非保留	
CHAR	非保留（不能是函数或类型）	保留	保留	保留
CHARACTER	非保留（不能是函数或类型）	保留	保留	保留
CHARACTERISTICS	非保留	非保留	非保留	
CHARACTERS	非保留	非保留		
CHARACTER_LENGTH	保留	保留	保留	
CHARACTER_SET_CATALOG	非保留	非保留	非保留	
CHARACTER_SET_NAME	非保留	非保留	非保留	
CHARACTER_SET_SCHEMA	非保留	非保留	非保留	
CHAR_LENGTH	保留	保留	保留	
CHECK	保留	保留	保留	保留
CHECKPOINT	非保留			
CLASS	非保留			
CLASS_ORIGIN	非保留	非保留	非保留	
CLOB	保留	保留		
CLOSE	非保留	保留	保留	保留
CLUSTER	非保留			
COALESCE	非保留（不能是函数或类型）	保留	保留	保留
COBOL	非保留	非保留	非保留	
COLLATE	保留	保留	保留	保留

COLLATION	保留（可以是函数或类型）	非保留	非保留	保留
COLLATION_CATALOG	非保留	非保留	非保留	
COLLATION_NAME	非保留	非保留	非保留	
COLLATION_SCHEMA	非保留	非保留	非保留	
COLLECT	保留	保留		
COLUMN	保留	保留	保留	保留
COLUMNS	非保留	非保留		
COLUMN_NAME	非保留	非保留	非保留	
COMMAND_FUNCTION	非保留	非保留	非保留	
COMMAND_FUNCTION_CODE	非保留	非保留		
COMMENT	非保留			
COMMENTS	非保留			
COMMIT	非保留	保留	保留	保留
COMMITTED	非保留	非保留	非保留	非保留
CONCURRENTLY	保留（可以是函数或类型）			
CONDITION	保留	保留		
CONDITION_NUMBER	非保留	非保留	非保留	
CONFIGURATION	非保留			
CONNECT	保留	保留	保留	
CONNECTION	非保留	非保留	非保留	保留
CONNECTION_NAME	非保留	非保留	非保留	
CONSTRAINT	保留	保留	保留	保留
CONSTRAINTS	非保留	非保留	非保留	保留
CONSTRAINT_CATALOG	非保留	非保留	非保留	
CONSTRAINT_NAME	非保留	非保留	非保留	
CONSTRAINT_SCHEMA	非保留	非保留	非保留	
CONSTRUCTOR	非保留	非保留		
CONTAINS	保留	非保留		
CONTENT	非保留	非保留	非保留	

CONTINUE	非保留	非保留	非保留	保留
CONTROL	非保留	非保留		
CONVERSION	非保留			
CONVERT	保留	保留	保留	
COPY	非保留			
CORR	保留	保留		
CORRESPONDING	保留	保留	保留	
COST	非保留			
COUNT	保留	保留	保留	
COVAR_POP	保留	保留		
COVAR_SAMP	保留	保留		
CREATE	保留	保留	保留	保留
CROSS	保留（可以是函数或类型）	保留	保留	保留
CSV	非保留			
CUBE	保留	保留		
CUME_DIST	保留	保留		
CURRENT	非保留	保留	保留	保留
CURRENT_CATALOG	保留	保留	保留	
CURRENT_DATE	保留	保留	保留	保留
CURRENT_DEFAULT_TRANSFORM_GROUP	保留	保留		
CURRENT_PATH	保留	保留		
CURRENT_ROLE	保留	保留	保留	
CURRENT_ROW	保留			
CURRENT_SCHEMA	保留（可以是函数或类型）	保留	保留	
CURRENT_TIME	保留	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留	保留
CURRENT_TRANSFORM_GROUP_FOR_TYPE	保留	保留		
CURRENT_USER	保留	保留	保留	保留
CURSOR	非保留	保留	保留	保留

CURSOR_NAME	非保留	非保留	非保留	
CYCLE	非保留	保留	保留	
DATA	非保留	非保留	非保留	非保留
DATABASE	非保留			
DATALINK	保留	保留		
DATE	保留	保留	保留	
DATETIME_INTERVAL_CODE	非保留	非保留	非保留	
DATETIME_INTERVAL_PRECISION	非保留	非保留	非保留	
DAY	非保留	保留	保留	保留
DB	非保留	非保留		
DEALLOCATE	非保留	保留	保留	保留
DEC	非保留（不能是函数或类型）	保留	保留	保留
DECIMAL	非保留（不能是函数或类型）	保留	保留	保留
DECLARE	非保留	保留	保留	保留
DEFAULT	保留	保留	保留	保留
DEFAULTS	非保留	非保留	非保留	
DEFERRABLE	保留	非保留	非保留	保留
DEFERRED	非保留	非保留	非保留	保留
DEFINED	非保留	非保留		
DEFINER	非保留	非保留	非保留	
DEGREE	非保留	非保留		
DELETE	非保留	保留	保留	保留
DELIMITER	非保留			
DELIMITERS	非保留			
DENSE_RANK	保留	保留		
DEPTH	非保留	非保留		
DEREF	保留	保留		
DERIVED	非保留	非保留		
DESC	保留	非保留	非保留	保留

DESCRIBE	保留	保留	保留	
DESCRIPTOR	非保留	非保留	保留	
DETERMINISTIC	保留	保留		
DIAGNOSTICS	非保留	非保留	保留	
DICTIONARY	非保留			
DISABLE	非保留			
DISCARD	非保留			
DISCONNECT	保留	保留	保留	
DISPATCH	非保留	非保留		
DISTINCT	保留	保留	保留	保留
DLNEWCOPY	保留	保留		
DLPREVIOUSCOPY	保留	保留		
DLURLCOMPLETE	保留	保留		
DLURLCOMPLETEONLY	保留	保留		
DLURLCOMPLETEWRITE	保留	保留		
DLURLPATH	保留	保留		
DLURLPATHONLY	保留	保留		
DLURLPATHWRITE	保留	保留		
DLURLSCHEME	保留	保留		
DLURLSERVER	保留	保留		
DLVALUE	保留	保留		
DO	保留			
DOCUMENT	非保留	非保留	非保留	
DOMAIN	非保留	非保留	非保留	保留
DOUBLE	非保留	保留	保留	保留
DROP	非保留	保留	保留	保留
DYNAMIC	保留	保留		
DYNAMIC_FUNCTION	非保留	非保留	非保留	
DYNAMIC_FUNCTION_CODE	非保留	非保留		
EACH	非保留	保留	保留	
ELEMENT	保留	保留		

`ELSE`	保留	保留	保留	保留
`EMPTY`	非保留	非保留		
`ENABLE`	非保留			
`ENCODING`	非保留	非保留	非保留	
`ENCRYPTED`	非保留			
`END`	保留	保留	保留	保留
`END-EXEC`	保留	保留	保留	
`END_FRAME`	保留			
`END_PARTITION`	保留			
`ENFORCED`	非保留			
`ENUM`	非保留			
`EQUALS`	保留	非保留		
`ESCAPE`	非保留	保留	保留	保留
`EVENT`	非保留			
`EVERY`	保留	保留		
`EXCEPT`	保留	保留	保留	保留
`EXCEPTION`	保留			
`EXCLUDE`	非保留	非保留	非保留	
`EXCLUDING`	非保留	非保留	非保留	
`EXCLUSIVE`	非保留			
`EXEC`	保留	保留	保留	
`EXECUTE`	非保留	保留	保留	保留
`EXISTS`	非保留（不能是函数或类型）	保留	保留	保留
`EXP`	非保留	非保留		
`EXPLAIN`	非保留			
`EXPRESSION`	非保留			
`EXTENSION`	非保留			
`EXTERNAL`	非保留	保留	保留	保留
`EXTRACT`	非保留（不能是函数或类型）	保留	保留	保留

FALSE	保留	保留	保留	保留
FAMILY	非保留			
FETCH	保留	保留	保留	保留
FILE	非保留	非保留		
FILTER	保留	保留		
FINAL	非保留	非保留		
FIRST	非保留	非保留	非保留	保留
FIRST_VALUE	保留	保留		
FLAG	非保留	非保留		
FLOAT	非保留（不能是函数或类型）	保留	保留	保留
FLOOR	保留	保留		
FOLLOWING	非保留	非保留	非保留	
FOR	保留	保留	保留	保留
FORCE	非保留			
FOREIGN	保留	保留	保留	保留
FORTRAN	非保留	非保留	非保留	
FORWARD	非保留			
FOUND	非保留	非保留	保留	
FRAME_ROW	保留			
FREE	保留	保留		
FREEZE	保留（可以是函数或类型）			
FROM	保留	保留	保留	保留
FS	非保留	非保留		
FULL	保留（可以是函数或类型）	保留	保留	保留
FUNCTION	非保留	保留	保留	
FUNCTIONS	非保留			
FUSION	保留	保留		
G	非保留	非保留		

GENERAL	非保留	非保留		
GENERATED	非保留	非保留		
GET	保留	保留	保留	
GLOBAL	非保留	保留	保留	保留
GO	非保留	非保留	保留	
GOTO	非保留	非保留	保留	
GRANT	保留	保留	保留	保留
GRANTED	非保留	非保留	非保留	
GREATEST	非保留（不能是函数或类型）			
GROUP	保留	保留	保留	保留
GROUPING	保留	保留		
GROUPS	保留			
HANDLER	非保留			
HAVING	保留	保留	保留	保留
HEADER	非保留			
HEX	非保留	非保留		
HIERARCHY	非保留	非保留		
HOLD	非保留	保留	保留	
HOUR	非保留	保留	保留	保留
ID	非保留	非保留		
IDENTITY	非保留	保留	保留	保留
IF	非保留			
IGNORE	非保留	非保留		
ILIKE	保留（可以是函数或类型）			
IMMEDIATE	非保留	非保留	非保留	保留
IMMEDIATELY	非保留			
IMMUTABLE	非保留			
IMPLEMENTATION	非保留	非保留		
IMPLICIT	非保留			

`IMPORT`	保留	保留		
`IN`	保留	保留	保留	保留
`INCLUDING`	非保留	非保留	非保留	
`INCREMENT`	非保留	非保留	非保留	
`INDENT`	非保留	非保留		
`INDEX`	非保留			
`INDEXES`	非保留			
`INDICATOR`	保留	保留	保留	
`INHERIT`	非保留			
`INHERITS`	非保留			
`INITIALLY`	保留	非保留	非保留	保留
`INLINE`	非保留			
`INNER`	保留（可以是函数或类型）	保留	保留	保留
`INOUT`	非保留（不能是函数或类型）	保留	保留	
`INPUT`	非保留	非保留	非保留	保留
`INSENSITIVE`	非保留	保留	保留	保留
`INSERT`	非保留	保留	保留	保留
`INSTANCE`	非保留	非保留		
`INSTANTIABLE`	非保留	非保留		
`INSTEAD`	非保留	非保留	非保留	
`INT`	非保留（不能是函数或类型）	保留	保留	保留
`INTEGER`	非保留（不能是函数或类型）	保留	保留	保留
`INTEGRITY`	非保留	非保留		
`INTERSECT`	保留	保留	保留	保留
`INTERSECTION`	保留	保留		
`INTERVAL`	非保留（不能是函数或类型）	保留	保留	保留

`INTO`	保留	保留	保留	保留
`INVOKER`	非保留	非保留	非保留	
`IS`	保留（可以是函数或类型）	保留	保留	保留
`ISNULL`	保留（可以是函数或类型）			
`ISOLATION`	非保留	非保留	非保留	保留
`JOIN`	保留（可以是函数或类型）	保留	保留	保留
`K`	非保留	非保留		
`KEY`	非保留	非保留	非保留	保留
`KEY_MEMBER`	非保留	非保留		
`KEY_TYPE`	非保留	非保留		
`LABEL`	非保留			
`LAG`	保留	保留		
`LANGUAGE`	非保留	保留	保留	保留
`LARGE`	非保留	保留	保留	
`LAST`	非保留	非保留	非保留	保留
`LAST_VALUE`	保留	保留		
`LATERAL`	保留	保留	保留	
`LC_COLLATE`	非保留			
`LC_CTYPE`	非保留			
`LEAD`	保留	保留		
`LEADING`	保留	保留	保留	保留
`LEAKPROOF`	非保留			
`LEAST`	非保留（不能是函数或类型）			
`LEFT`	保留（不能是函数或类型）	保留	保留	保留
`LENGTH`	非保留	非保留	非保留	

LEVEL	非保留	非保留	非保留	保留
LIBRARY	非保留	非保留		
LIKE	保留（可以是函数或类型）	保留	保留	保留
LIKE_REGEX	保留	保留		
LIMIT	保留	非保留	非保留	
LINK	非保留	非保留		
LISTEN	非保留			
LN	保留	保留		
LOAD	非保留			
LOCAL	非保留	保留	保留	保留
LOCALTIME	保留	保留	保留	
LOCALTIMESTAMP	保留	保留	保留	
LOCATION	非保留	非保留	非保留	
LOCATOR	非保留	非保留		
LOCK	非保留			
LOWER	保留	保留	保留	
M	非保留	非保留		
MAP	非保留	非保留		
MAPPING	非保留	非保留	非保留	
MATCH	非保留	保留	保留	保留
MATCHED	非保留	非保留		
MATERIALIZED	非保留			
MAX	保留	保留	保留	
MAXVALUE	非保留	非保留	非保留	
MAX_CARDINALITY	保留			
MEMBER	保留	保留		
MERGE	保留	保留		
MESSAGE_LENGTH	非保留	非保留	非保留	
MESSAGE_OCTET_LENGTH	非保留	非保留	非保留	
MESSAGE_TEXT	非保留	非保留	非保留	

METHOD	保留	保留		
MIN	保留	保留	保留	
MINUTE	非保留	保留	保留	保留
MINVALUE	非保留	非保留	非保留	
MOD	保留	保留		
MODE	非保留			
MODIFIES	保留	保留		
MODULE	保留	保留	保留	
MONTH	非保留	保留	保留	保留
MORE	非保留	非保留	非保留	
MOVE	非保留			
MULTISET	保留	保留		
MUMPS	非保留	非保留	非保留	
NAME	非保留	非保留	非保留	非保留
NAMES	非保留	非保留	非保留	保留
NAMESPACE	非保留	非保留		
NATIONAL	非保留（不能是函数或类型）	保留	保留	保留
NATURAL	保留（可以是函数或类型）	保留	保留	保留
NCHAR	非保留（不能是函数或类型）	保留	保留	保留
NCLOB	保留	保留		
NESTING	非保留	非保留		
NEW	保留	保留		
NEXT	非保留	非保留	非保留	保留
NFC	非保留	非保留		
NFD	非保留	非保留		
NFKC	非保留	非保留		
NFKD	非保留	非保留		

`NIL`	非保留	非保留		
`NO`	非保留	保留	保留	保留
`NONE`	非保留（不能是函数或类型）	保留	保留	
`NORMALIZE`	保留	保留		
`NORMALIZED`	非保留	非保留		
`NOT`	保留	保留	保留	保留
`NOTHING`	非保留			
`NOTIFY`	非保留			
`NOTNULL`	保留（可以是函数或类型）			
`NOWAIT`	非保留			
`NTH_VALUE`	保留	保留		
`NTILE`	保留	保留		
`NULL`	保留	保留	保留	保留
`NULLABLE`	非保留	非保留	非保留	
`NULLIF`	非保留（不能是函数或类型）	保留	保留	保留
`NULLS`	非保留	非保留	非保留	
`NUMBER`	非保留	非保留	非保留	
`NUMERIC`	非保留（不能是函数或类型）	保留	保留	保留
`OBJECT`	非保留	非保留	非保留	
`OCCURRENCES_REGEX`	保留	保留		
`OCTETS`	非保留	非保留		
`OCTET_LENGTH`	保留	保留	保留	
`OF`	非保留	保留	保留	保留
`OFF`	非保留	非保留	非保留	
`OFFSET`	保留	保留	保留	
`OIDS`	非保留			
`OLD`	保留	保留		

ON	保留	保留	保留	保留
ONLY	保留	保留	保留	保留
OPEN	保留	保留	保留	
OPERATOR	非保留			
OPTION	非保留	非保留	非保留	保留
OPTIONS	非保留	非保留	非保留	
OR	保留	保留	保留	保留
ORDER	保留	保留	保留	保留
ORDERING	非保留	非保留		
ORDINALITY	非保留	非保留		
OTHERS	非保留	非保留		
OUT	非保留（不能是函数或类型）	保留	保留	
OUTER	保留（可以是函数或类型）	保留	保留	保留
OUTPUT	非保留	非保留	保留	
OVER	保留（可以是函数或类型）	保留	保留	
OVERLAPS	保留（可以是函数或类型）	保留	保留	保留
OVERLAY	非保留（不能是函数或类型）	保留	保留	
OVERRIDING	非保留	非保留		
OWNED	非保留			
OWNER	非保留			
P	非保留	非保留		
PAD	非保留	非保留	保留	
PARAMETER	保留	保留		
PARAMETER_MODE	非保留	非保留		
PARAMETER_NAME	非保留	非保留		
PARAMETER_ORDINAL_POSITION	非保留	非保留		

PARAMETER_SPECIFIC_CATALOG	非保留	非保留		
PARAMETER_SPECIFIC_NAME	非保留	非保留		
PARAMETER_SPECIFIC_SCHEMA	非保留	非保留		
PARSER	非保留			
PARTIAL	非保留	非保留	非保留	保留
PARTITION	非保留	保留	保留	
PASCAL	非保留	非保留	非保留	
PASSING	非保留	非保留	非保留	
PASSTHROUGH	非保留	非保留		
PASSWORD	非保留			
PATH	非保留	非保留		
PERCENT	保留			
PERCENTILE_CONT	保留	保留		
PERCENTILE_DISC	保留	保留		
PERCENT_RANK	保留	保留		
PERIOD	保留			
PERMISSION	非保留	非保留		
PLACING	保留	非保留	非保留	
PLANS	非保留			
PLI	非保留	非保留	非保留	
PORTION	保留			
POSITION	非保留（不能是函数或类型）	保留	保留	保留
POSITION_REGEX	保留	保留		
POWER	保留	保留		
PRECEDES	保留			
PRECEDING	非保留	非保留	非保留	
PRECISION	非保留（不能是函数或类型）	保留	保留	保留
PREPARE	非保留	保留	保留	保留
PREPARED	非保留			

PRESERVE	非保留	非保留	非保留	保留
PRIMARY	保留	保留	保留	保留
PRIOR	非保留	非保留	非保留	保留
PRIVILEGES	非保留	非保留	非保留	保留
PROCEDURAL	非保留			
PROCEDURE	非保留	保留	保留	保留
PROGRAM	非保留			
PUBLIC	非保留	非保留	保留	
QUOTE	非保留			
RANGE	非保留	保留	保留	
RANK	保留	保留		
READ	非保留	非保留	非保留	保留
READS	保留	保留		
REAL	非保留（不能是函数或类型）	保留	保留	保留
REASSIGN	非保留			
RECHECK	非保留			
RECOVERY	非保留	非保留		
RECURSIVE	非保留	保留	保留	
REF	非保留	保留	保留	
REFERENCES	保留	保留	保留	保留
REFERENCING	保留	保留		
REFRESH	非保留			
REGR_AVGX	保留	保留		
REGR_AVGY	保留	保留		
REGR_COUNT	保留	保留		
REGR_INTERCEPT	保留	保留		
REGR_R2	保留	保留		
REGR_SLOPE	保留	保留		
REGR_SXX	保留	保留		
REGR_SXY	保留	保留		

REGR_SYY	保留	保留		
REINDEX	非保留			
RELATIVE	非保留	非保留	非保留	保留
RELEASE	非保留	保留	保留	
RENAME	非保留			
REPEATABLE	非保留	非保留	非保留	非保留
REPLACE	非保留			
REPLICA	非保留			
REQUIRING	非保留	非保留		
RESET	非保留			
RESPECT	非保留	非保留		
RESTART	非保留	非保留	非保留	
RESTORE	非保留	非保留		
RESTRICT	非保留	非保留	非保留	保留
RESULT	保留	保留		
RETURN	保留	保留		
RETURNED_CARDINALITY	保留	保留		
RETURNED_LENGTH	非保留	非保留	非保留	
RETURNED_OCTET_LENGTH	非保留	非保留	非保留	
RETURNED_SQLSTATE	非保留	非保留	非保留	
RETURNING	保留	非保留	非保留	
RETURNS	非保留	保留	保留	
REVOKE	非保留	保留	保留	保留
RIGHT	保留（可以是函数或类型）	保留	保留	保留
ROLE	非保留	非保留	非保留	
ROLLBACK	非保留	保留	保留	保留
ROLLUP	保留	保留		
ROUTINE	非保留	非保留		
ROUTINE_CATALOG	非保留	非保留		
ROUTINE_NAME	非保留	非保留		

ROUTINE_SCHEMA	非保留	非保留		
ROW	非保留（不能是函数或类型）	保留	保留	
ROWS	非保留	保留	保留	保留
ROW_COUNT	非保留	非保留	非保留	
ROW_NUMBER	保留	保留		
RULE	非保留			
SAVEPOINT	非保留	保留	保留	
SCALE	非保留	非保留	非保留	
SCHEMA	非保留	非保留	非保留	保留
SCHEMA_NAME	非保留	非保留	非保留	
SCOPE	保留	保留		
SCOPE_CATALOG	非保留	非保留		
SCOPE_NAME	非保留	非保留		
SCOPE_SCHEMA	非保留	非保留		
SCROLL	非保留	保留	保留	保留
SEARCH	非保留	保留	保留	
SECOND	非保留	保留	保留	保留
SECTION	非保留	非保留	保留	
SECURITY	非保留	非保留	非保留	
SELECT	保留	保留	保留	保留
SELECTIVE	非保留	非保留		
SELF	非保留	非保留		
SENSITIVE	保留	保留		
SEQUENCE	非保留	非保留	非保留	
SEQUENCES	非保留			
SERIALIZABLE	非保留	非保留	非保留	非保留
SERVER	非保留	非保留	非保留	
SERVER_NAME	非保留	非保留	非保留	
SESSION	非保留	非保留	非保留	保留
SESSION_USER	保留	保留	保留	保留

SET	非保留	保留	保留	保留
SETOF	非保留（不能是函数或类型）			
SETS	非保留	非保留		
SHARE	非保留			
SHOW	非保留			
SIMILAR	保留（可以是函数或类型）	保留	保留	
SIMPLE	非保留	非保留	非保留	
SIZE	非保留	非保留	保留	
SMALLINT	非保留（不能是函数或类型）	保留	保留	保留
SNAPSHOT	非保留			
SOME	保留	保留	保留	保留
SOURCE	非保留	非保留		
SPACE	非保留	非保留	保留	
SPECIFIC	保留	保留		
SPECIFICTYPE	保留	保留		
SPECIFIC_NAME	非保留	非保留		
SQL	保留	保留	保留	
SQLCODE	保留			
SQLERROR	保留			
SQLEXCEPTION	保留	保留		
SQLSTATE	保留	保留	保留	
SQLWARNING	保留	保留		
SQRT	保留	保留		
STABLE	非保留			
STANDALONE	非保留	非保留	非保留	
START	非保留	保留	保留	
STATE	非保留	非保留		
STATEMENT	非保留	非保留	非保留	

STATIC	保留	保留		
STATISTICS	非保留			
STDDEV_POP	保留	保留		
STDDEV_SAMP	保留	保留		
STDIN	非保留			
STDOUT	非保留			
STORAGE	非保留			
STRICT	非保留			
STRIP	非保留	非保留	非保留	
STRUCTURE	非保留	非保留		
STYLE	非保留	非保留		
SUBCLASS_ORIGIN	非保留	非保留	非保留	
SUBMULTISET	保留	保留		
SUBSTRING	非保留（不能是函数或类型）	保留	保留	保留
SUBSTRING_REGEX	保留	保留		
SUCCEEDS	保留			
SUM	保留	保留	保留	
SYMMETRIC	保留	保留	保留	
SYSID	非保留			
SYSTEM	非保留	保留	保留	
SYSTEM_TIME	保留			
SYSTEM_USER	保留	保留	保留	
T	非保留	非保留		
TABLE	保留	保留	保留	保留
TABLES	非保留			
TABLESAMPLE	保留	保留		
TABLESPACE	非保留			
TABLE_NAME	非保留	非保留	非保留	
TEMP	非保留			
TEMPLATE	非保留			

TEMPORARY	非保留	非保留	非保留	保留
TEXT	非保留			
THEN	保留	保留	保留	保留
TIES	非保留	非保留		
TIME	非保留（不能是函数或类型）	保留	保留	保留
TIMESTAMP	非保留（不能是函数或类型）	保留	保留	保留
TIMEZONE_HOUR	保留	保留	保留	
TIMEZONE_MINUTE	保留	保留	保留	
TO	保留	保留	保留	保留
TOKEN	非保留	非保留		
TOP_LEVEL_COUNT	非保留	非保留		
TRAILING	保留	保留	保留	保留
TRANSACTION	非保留	非保留	非保留	保留
TRANSACTIONS_COMMITTED	非保留	非保留		
TRANSACTIONS_ROLLED_BACK	非保留	非保留		
TRANSACTION_ACTIVE	非保留	非保留		
TRANSFORM	非保留	非保留		
TRANSFORMS	非保留	非保留		
TRANSLATE	保留	保留	保留	
TRANSLATE_REGEX	保留	保留		
TRANSLATION	保留	保留	保留	
TREAT	非保留（不能是函数或类型）	保留	保留	
TRIGGER	非保留	保留	保留	
TRIGGER_CATALOG	非保留	非保留		
TRIGGER_NAME	非保留	非保留		
TRIGGER_SCHEMA	非保留	非保留		
TRIM	非保留（不能是函数或类型）	保留	保留	保留

TRIM_ARRAY	保留	保留		
TRUE	保留	保留	保留	保留
TRUNCATE	非保留	保留	保留	
TRUSTED	非保留			
TYPE	非保留	非保留	非保留	非保留
TYPES	非保留			
UESCAPE	保留	保留		
UNBOUNDED	非保留	非保留	非保留	
UNCOMMITTED	非保留	非保留	非保留	非保留
UNDER	非保留	非保留		
UNENCRYPTED	非保留			
UNION	保留	保留	保留	保留
UNIQUE	保留	保留	保留	保留
UNKNOWN	非保留	保留	保留	保留
UNLINK	非保留	非保留		
UNLISTEN	非保留			
UNLOGGED	非保留			
UNNAMED	非保留	非保留	非保留	
UNNEST	保留	保留		
UNTIL	非保留			
UNTYPED	非保留	非保留		
UPDATE	非保留	保留	保留	保留
UPPER	保留	保留	保留	
URI	非保留	非保留		
USAGE	非保留	非保留	保留	
USER	保留	保留	保留	保留
USER_DEFINED_TYPE_CATALOG	非保留	非保留		
USER_DEFINED_TYPE_CODE	非保留	非保留		
USER_DEFINED_TYPE_NAME	非保留	非保留		
USER_DEFINED_TYPE_SCHEMA	非保留	非保留		

USING	保留	保留	保留	保留
VACUUM	非保留			
VALID	非保留	非保留	非保留	
VALIDATE	非保留			
VALIDATOR	非保留			
VALUE	非保留	保留	保留	保留
VALUES	非保留（不能是函数或类型）	保留	保留	保留
VALUE_OF	保留			
VARBINARY	保留	保留		
VARCHAR	非保留（不能是函数或类型）	保留	保留	保留
VARIADIC	保留			
VARYING	非保留	保留	保留	保留
VAR_POP	保留	保留		
VAR_SAMP	保留	保留		
VERBOSE	保留（不能是函数或类型）			
VERSION	非保留	非保留	非保留	
VERSIONING	保留			
VIEW	非保留	非保留	非保留	保留
VOLATILE	非保留			
WHEN	保留	保留	保留	保留
WHENEVER	保留	保留	保留	
WHERE	保留	保留	保留	保留
WHITESPACE	非保留	非保留	非保留	
WIDTH_BUCKET	保留	保留		
WINDOW	保留	保留	保留	
WITH	保留	保留	保留	保留
WITHIN	保留	保留		
WITHOUT	非保留	保留	保留	

WORK	非保留	非保留	非保留	保留
WRAPPER	非保留	非保留	非保留	
WRITE	非保留	非保留	非保留	保留
XML	非保留	保留	保留	
XMLAGG	保留	保留		
XMLATTRIBUTES	非保留（不能是函数或类型）	保留	保留	
XMLBINARY	保留	保留		
XMLCAST	保留	保留		
XMLCOMMENT	保留	保留		
XMLCONCAT	非保留（不能是函数或类型）	保留	保留	
XMLDECLARATION	非保留	非保留		
XMLDOCUMENT	保留	保留		
XMLELEMENT	非保留（不能是函数或类型）	保留	保留	
XMLEXISTS	非保留（不能是函数或类型）	保留	保留	
XMLFOREST	非保留（不能是函数或类型）	保留	保留	
XMLITERATE	保留	保留		
XMLNAMESPACES	保留	保留		
XMLPARSE	非保留（不能是函数或类型）	保留	保留	
XMLPI	非保留（不能是函数或类型）	保留	保留	
XMLQUERY	保留	保留		
XMLROOT	非保留（不能是函数或类型）			
XMLSCHEMA	非保留	非保留		

XMLSERIALIZE	非保留（不能是函数或类型）	保留	保留	
XMLTABLE	保留	保留		
XMLTEXT	保留	保留		
XMLVALIDATE	保留	保留		
YEAR	非保留	保留	保留	保留
YES	非保留	非保留	非保留	
ZONE	非保留	非保留	非保留	保留

Appendix D. SQL兼容性

Table of Contents

本节试图描述PostgreSQL在多大程度上遵循SQL标准。 下面的信息不是兼容性的全部内容，但是它提供了一个从用户角度来看， 既合理又有用的足够的细节信息。

SQL标准的正式名称是ISO/IEC 9075 "Database Language SQL"。 标准的修改版会经常地发布； 最近更新的一个版本出现在2011年。 那个版本被称作ISO/IEC 9075:2011， 或者简称SQL:2011。 这个版本之前的是SQL:2008, SQL:2003, SQL:1999和SQL-92。 每个版本都替代前面那个， 所以声称兼容早期版本没有什么官方的好处。 PostgreSQL开发瞄准兼容标准最新官方版本， 只要这样的兼容不会与传统的特性或者常识冲突。 许多SQL标准要求的特性都得到了支持， 只是有些时候语法或者函数略微不同。 更多有关标准兼容的特性将在未来的版本里看到。

SQL-92为兼容性定义了三种特性集合： Entry, Intermediate和Full。大部分数据库管理系统声明在Entry 级别遵循SQL标准兼容性， 因为中等和完全的特性要么是太庞大， 要么就是和传统的行为相冲突。

从SQL:1999开始，SQL标准定义了一个很大的独立特性集合， 而不是SQL-92那样宽泛而又低效率的三个级别。 这些特性中的一个很大的子集形成"核心"特性， 它们是每种兼容SQL的实现必须提供的特性。 其它的特性都是可选的。有些可选的特性组合在一起形成"包"， SQL的实现可以号称遵循这些包， 也就是声称遵循特定的特性组。

SQL:2003标准也分裂成一系列部分： 每种都用一个缩写来标识。请注意这些部分并非连续编号的。

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 基础(SQL/Foundation)
- ISO/IEC 9075-3调用层接口(SQL/CLI)
- ISO/IEC 9075-4永久存储模块(SQL/PSM)
- ISO/IEC 9075-9外部数据管理(SQL/MED)
- ISO/IEC 9075-10对象语言绑定(SQL/OLB)
- ISO/IEC 9075-11信息及定义模式(SQL/Schemata)
- ISO/IEC 9075-13 Java语言的过程和类型(SQL/JRT)

- ISO/IEC 9075-14 XML相关的规范(SQL/XML)

PostgreSQL涵盖1,2,9,11和14。3被ODBC驱动覆盖， 13被PL/Java插件覆盖。但是这些组件目前没有严格的兼容检查。 目前没有实现PostgreSQL的第4和10部分。

PostgreSQL支持大多数SQL:2011的主要特性。 在总共179个强制要求完全兼容的核心特性里， PostgreSQL遵循至少160个。另外， PostgreSQL还支持一长串可选的特性。 值得一提的是，在写这些的时候， 没有任何当前版本的数据库管理系统声称支持全部核心SQL:2011。

下面的两节列出了PostgreSQL支持的特性， 以及PostgreSQL目前尚不支持的SQL:2011特性。 这两个列表都是近似的： 被列为支持的特性可能在某些次要细节方面与标准不一致， 被列为不支持的特性也可能实际上已经被实现。 文档的主体部分包含了大多数能否正常工作的精确信息。

> **Note:** 包含一个连字符的特性代码表示一个子特性。 因此，如果不支持特定的子特性，那么主特性也会列在不支持的特性， 即使支持其它的子特性也如此。

D.1. 支持的特性

标识符	包	描述	注释
B012	Embedded C		
B021	Direct SQL		
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims tra spaces CHARA values b countin
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	

E021-08	Core	UPPER and LOWER functions
E021-09	Core	TRIM function
E021-10	Core	Implicit casting among the character string types
E021-11	Core	POSITION function
E021-12	Core	Character comparison
E031	Core	Identifiers
E031-01	Core	Delimited identifiers
E031-02	Core	Lower case identifiers
E031-03	Core	Trailing underscore
E051	Core	Basic query specification
E051-01	Core	SELECT DISTINCT
E051-02	Core	GROUP BY clause
E051-04	Core	GROUP BY can contain columns not in <select list>
E051-05	Core	Select list items can be renamed
E051-06	Core	HAVING clause
E051-07	Core	Qualified * in select list
E051-08	Core	Correlation names in the FROM clause
E051-09	Core	Rename columns in the FROM clause
E061	Core	Basic predicates and search conditions
E061-01	Core	Comparison predicate
E061-	Core	BETWEEN predicate

02	Core	BETWEEN predicate
E061-03	Core	IN predicate with list of values
E061-04	Core	LIKE predicate
E061-05	Core	LIKE predicate ESCAPE clause
E061-06	Core	NULL predicate
E061-07	Core	Quantified comparison predicate
E061-08	Core	EXISTS predicate
E061-09	Core	Subqueries in comparison predicate
E061-11	Core	Subqueries in IN predicate
E061-12	Core	Subqueries in quantified comparison predicate
E061-13	Core	Correlated subqueries
E061-14	Core	Search condition
E071	Core	Basic query expressions
E071-01	Core	UNION DISTINCT table operator
E071-02	Core	UNION ALL table operator
E071-03	Core	EXCEPT DISTINCT table operator
E071-05	Core	Columns combined via table operators need not have exactly the same data type
E071-06	Core	Table operators in subqueries
E081	Core	Basic Privileges
E081-01	Core	SELECT privilege

02		
E081-03	Core	INSERT privilege at the table level
E081-04	Core	UPDATE privilege at the table level
E081-05	Core	UPDATE privilege at the column level
E081-06	Core	REFERENCES privilege at the table level
E081-07	Core	REFERENCES privilege at the column level
E081-08	Core	WITH GRANT OPTION
E081-09	Core	USAGE privilege
E081-10	Core	EXECUTE privilege
E091	Core	Set functions
E091-01	Core	AVG
E091-02	Core	COUNT
E091-03	Core	MAX
E091-04	Core	MIN
E091-05	Core	SUM
E091-06	Core	ALL quantifier
E091-07	Core	DISTINCT quantifier
E101	Core	Basic data manipulation
E101-01	Core	INSERT statement
E101-03	Core	Searched UPDATE statement
E101-04	Core	Searched DELETE statement

04		
E111	Core	Single row SELECT statement
E121	Core	Basic cursor support
E121-01	Core	DECLARE CURSOR
E121-02	Core	ORDER BY columns need not be in select list
E121-03	Core	Value expressions in ORDER BY clause
E121-04	Core	OPEN statement
E121-06	Core	Positioned UPDATE statement
E121-07	Core	Positioned DELETE statement
E121-08	Core	CLOSE statement
E121-10	Core	FETCH statement implicit NEXT
E121-17	Core	WITH HOLD cursors
E131	Core	Null value support (nulls in lieu of values)
E141	Core	Basic integrity constraints
E141-01	Core	NOT NULL constraints
E141-02	Core	UNIQUE constraints of NOT NULL columns
E141-03	Core	PRIMARY KEY constraints
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action
E141-06	Core	CHECK constraints
E141-07	Core	Column defaults
E141-	Core	NOT NULL inferred on

E141-08	Core	NOT NULL inferred on PRIMARY KEY
E141-10	Core	Names in a foreign key can be specified in any order
E151	Core	Transaction support
E151-01	Core	COMMIT statement
E151-02	Core	ROLLBACK statement
E152	Core	Basic SET TRANSACTION statement
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses
E153	Core	Updatable queries with subqueries
E161	Core	SQL comments using leading double minus
E171	Core	SQLSTATE support
F021	Core	Basic information schema
F021-01	Core	COLUMNS view
F021-02	Core	TABLES view
F021-03	Core	VIEWS view
F021-04	Core	TABLE_CONSTRAINTS view
F021-05	Core	REFERENTIAL_CONSTRAINTS view
F021-06	Core	CHECK_CONSTRAINTS view
F031	Core	Basic schema manipulation
F031-01	Core	CREATE TABLE statement to create persistent base tables
F031-	Core	CREATE VIEW statement

02	Core	CREATE VIEW statement
F031-03	Core	GRANT statement
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause
F031-13	Core	DROP TABLE statement: RESTRICT clause
F031-16	Core	DROP VIEW statement: RESTRICT clause
F031-19	Core	REVOKE statement: RESTRICT clause
F032	CASCADE drop behavior	
F033	ALTER TABLE statement: DROP COLUMN clause	
F034	Extended REVOKE statement	
F034-01	REVOKE statement performed by other than the owner of a schema object	
F034-02	REVOKE statement: GRANT OPTION FOR clause	
F034-03	REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table
F041-01	Core	Inner join (but not necessarily the INNER keyword)
F041-02	Core	INNER keyword
F041-03	Core	LEFT OUTER JOIN
F041-04	Core	RIGHT OUTER JOIN
F041-05	Core	Outer joins can be nested
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join
F041-08	Core	All comparison operators are supported (rather than just =)

F051-01	Core	DATE data type (including support of DATE literal)
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types
F051-05	Core	Explicit CAST between datetime types and character string types
F051-06	Core	CURRENT_DATE
F051-07	Core	LOCALTIME
F051-08	Core	LOCALTIMESTAMP
F052	Enhanced datetime facilities	Intervals and datetime arithmetic
F053	OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views
F111	Isolation levels other than SERIALIZABLE	
F111-01	READ UNCOMMITTED isolation level	
F111-02	READ COMMITTED isolation level	
F111-03	REPEATABLE READ isolation level	
F131	Core	Grouped operations
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views
F131-02	Core	Multiple tables supported in queries with grouped views
F131-03	Core	Set functions supported in queries with grouped views

03		queries with grouped views
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views
F171	Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions
F200	TRUNCATE TABLE statement	
F201	Core	CAST function
F221	Core	Explicit defaults
F222	INSERT statement: DEFAULT VALUES clause	
F231	Privilege tables	
F231-01	TABLE_PRIVILEGES view	
F231-02	COLUMN_PRIVILEGES view	
F231-03	USAGE_PRIVILEGES view	
F251	Domain support	
F261	Core	CASE expression
F261-01	Core	Simple CASE
F261-02	Core	Searched CASE
F261-03	Core	NULLIF
F261-04	Core	COALESCE
F262	Extended CASE expression	
F271	Compound character literals	
F281	LIKE enhancements	
F302	INTERSECT table operator	
F302-	INTERSECT DISTINCT table	

F302-01	INTERSECT DISTINCT table operator	
F302-02	INTERSECT ALL table operator	
F304	EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA
F311-02	Core	CREATE TABLE for persistent base tables
F311-03	Core	CREATE VIEW
F311-05	Core	GRANT statement
F321	User authorization	
F361	Subprogram support	
F381	Extended schema manipulation	
F381-01	ALTER TABLE statement: ALTER COLUMN clause	
F381-02	ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03	ALTER TABLE statement: DROP CONSTRAINT clause	
F382	Alter column data type	
F383	Set column not null clause	
F391	Long identifiers	
F392	Unicode escapes in identifiers	
F393	Unicode escapes in literals	
F401	Extended joined table	
F401-01	NATURAL JOIN	
F401-02	FULL OUTER JOIN	
F401-04	CROSS JOIN	
F402	Named column joins for LOBs, arrays, and multisets	

F411	Enhanced datetime facilities	Time zone specification	regardir literal interpre
F421	National character		
F431	Read-only scrollable cursors		
F431-01	FETCH with explicit NEXT		
F431-02	FETCH FIRST		
F431-03	FETCH LAST		
F431-04	FETCH PRIOR		
F431-05	FETCH ABSOLUTE		
F431-06	FETCH RELATIVE		
F441	Extended set function support		
F442	Mixed column references in set functions		
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502	Enhanced documentation tables		
F502-01	SQL_SIZING_PROFILES view		
F502-02	SQL_IMPLEMENTATION_INFO view		
F502-			

F531	Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision
F561	Full value expressions	
F571	Truth value tests	
F591	Derived tables	
F611	Indicator data types	
F641	Row and table constructors	
F651	Catalog name qualifiers	
F661	Simple tables	
F672	Retrospective check constraints	
F690	Collation support	but no character set support
F692	Extended collation support	
F701	Enhanced integrity management	Referential update actions
F711	ALTER domain	
F731	INSERT column privileges	
F761	Session management	
F762	CURRENT_CATALOG	
F763	CURRENT_SCHEMA	
F771	Connection management	
F781	Self-referencing operations	
F791	Insensitive cursors	
F801	Full set function	
F850	Top-level <order by clause> in <query expression>	
F851	<order by clause> in subqueries	
F852	Top-level <order by clause> in views	
F855	Nested <order by clause> in <query expression>	
F856	Nested <fetch first clause> in <query expression>	
F857	Top-level <fetch first clause> in <query expression>	

	<query expression>	
F858	<fetch first clause> in subqueries	
F859	Top-level <fetch first clause> in views	
F860	<fetch first row count> in <fetch first clause>	
F861	Top-level <result offset clause> in <query expression>	
F862	<result offset clause> in subqueries	
F863	Nested <result offset clause> in <query expression>	
F864	Top-level <result offset clause> in views	
F865	<offset row count> in <result offset clause>	
S071	Enhanced object support	SQL paths in function and type name resolution
S092	Arrays of user-defined types	
S095	Array constructors by query	
S096	Optional array bounds	
S098	ARRAY_AGG	
S111	Enhanced object support	ONLY in query expressions
S201	SQL-invoked routines on arrays	
S201-01	Array parameters	
S201-02	Array as result type of functions	
S211	Enhanced object support	User-defined cast functions
T031	BOOLEAN data type	
T071	BIGINT data type	
T121	WITH (excluding RECURSIVE) in query expression	
T122	WITH (excluding RECURSIVE) in subquery	
T131	Recursive query	

T131	Recursive query	
T132	Recursive query in subquery	
T141	SIMILAR predicate	
T151	DISTINCT predicate	
T152	DISTINCT predicate with negation	
T171	LIKE clause in table definition	
T172	AS subquery clause in table definition	
T173	Extended LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT
T201	Enhanced integrity management	Comparable data types for referential constraints
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table
T211-02	Active database, Enhanced integrity management	BEFORE triggers
T211-03	Active database, Enhanced integrity management	AFTER triggers
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked
T211-07	Active database, Enhanced integrity management	TRIGGER privilege
T212	Enhanced integrity management	Enhanced trigger capability
T213	INSTEAD OF triggers	
T231	Sensitive cursors	
T241	START TRANSACTION statement	
T271	Savepoints	
T281	SELECT privilege with column granularity	

T312	OVERLAY function	
T321-01	Core	User-defined functions with no overloading
T321-03	Core	Function invocation
T321-06	Core	ROUTINES view
T321-07	Core	PARAMETERS view
T323	Explicit security for external routines	
T331	Basic roles	
T341	Overloading of SQL-invoked functions and procedures	
T351	Bracketed SQL comments (/.../ comments)	
T441	ABS and MOD functions	
T461	Symmetric BETWEEN predicate	
T491	LATERAL derived table	
T501	Enhanced EXISTS predicate	
T551	Optional key words for default syntax	
T581	Regular expression substring function	
T591	UNIQUE constraints of possibly null columns	
T614	NTILE function	
T615	LEAD and LAG functions	
T617	FIRST_VALUE and LAST_VALUE function	
T621	Enhanced numeric functions	
T631	Core	IN predicate with one list element
T651	SQL-schema statements in SQL routines	
T655	Cyclically dependent routines	

X011	Arrays of XML type
X016	Persistent XML values
X020	XMLConcat
X031	XMLElement
X032	XMLForest
X034	XMLAgg
X035	XMLAgg: ORDER BY option
X036	XMLComment
X037	XMLPI
X040	Basic table mapping
X041	Basic table mapping: nulls absent
X042	Basic table mapping: null as nil
X043	Basic table mapping: table as forest
X044	Basic table mapping: table as element
X045	Basic table mapping: with target namespace
X046	Basic table mapping: data mapping
X047	Basic table mapping: metadata mapping
X048	Basic table mapping: base64 encoding of binary strings
X049	Basic table mapping: hex encoding of binary strings
X050	Advanced table mapping
X051	Advanced table mapping: nulls absent
X052	Advanced table mapping: null as nil
X053	Advanced table mapping: table as forest
X054	Advanced table mapping: table as element
	Advanced table mapping: target

X055	namespace
X056	Advanced table mapping: data mapping
X057	Advanced table mapping: metadata mapping
X058	Advanced table mapping: base64 encoding of binary strings
X059	Advanced table mapping: hex encoding of binary strings
X060	XMLParse: Character string input and CONTENT option
X061	XMLParse: Character string input and DOCUMENT option
X070	XMLSerialize: Character string serialization and CONTENT option
X071	XMLSerialize: Character string serialization and DOCUMENT option
X072	XMLSerialize: Character string serialization
X090	XML document predicate
X120	XML parameters in SQL routines
X121	XML parameters in external routines
X400	Name and identifier mapping
X410	Alter column data type: XML type

D.2. 不支持的特性

下面这些在SQL:2011中定义的特性 在目前的PostgreSQL版本中还没有实现， 不过在某些情况下可以获得等效的功能。

标识符	包	描述	注释
B011	Embedded Ada		
B013	Embedded COBOL		
B014	Embedded Fortran		
B015	Embedded MUMPS		
B016	Embedded Pascal		
B017	Embedded PL/I		
B031	Basic dynamic SQL		
B032	Extended dynamic SQL		
B032-01	<describe input statement>		
B033	Untyped SQL-invoked function arguments		
B034	Dynamic specification of cursor attributes		
B035	Non-extended descriptor names		
B041	Extensions to embedded SQL exception declarations		
B051	Enhanced execution rights		
B111	Module language Ada		
B112	Module language C		
B113	Module language COBOL		
B114	Module language Fortran		
B115	Module language MUMPS		
B116	Module language Pascal		
B117	Module language PL/I		
B121	Routine language Ada		
B122	Routine language C		

B123	Routine language COBOL	
B124	Routine language Fortran	
B125	Routine language MUMPS	
B126	Routine language Pascal	
B127	Routine language PL/I	
B128	Routine language SQL	
B211	Module language Ada: VARCHAR and NUMERIC support	
B221	Routine language Ada: VARCHAR and NUMERIC support	
E182	Core	Module language
F054	TIMESTAMP in DATE type precedence list	
F121	Basic diagnostics management	
F121-01	GET DIAGNOSTICS statement	
F121-02	SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122	Enhanced diagnostics management	
F123	All diagnostics	
F181	Core	Multiple module support
F202	TRUNCATE TABLE: identity column restart option	
F263	Comma-separated predicates in simple CASE expression	
F291	UNIQUE predicate	
F301	CORRESPONDING in query expressions	
F311	Core	Schema definition statement
F311-04	Core	CREATE VIEW: WITH CHECK OPTION
F312	MERGE statement	
F313	Enhanced MERGE statement	
	MERGE statement with DELETE	

	branch		
F341	Usage tables	no ROUTINE_*_USAGE tables	
F384	Drop identity property clause		
F385	Drop column generation expression clause		
F386	Set identity column generation clause		
F394	Optional normal form specification		
F403	Partitioned joined tables		
F451	Character set definition		
F461	Named character sets		
F492	Optional table constraint enforcement		
F521	Enhanced integrity management	Assertions	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F693	SQL-session and client module collations		
F695	Translation support		
F696	Additional translation documentation		
F721	Deferrable constraints	foreign and unique keys only	
F741	Referential MATCH types	no partial match yet	
F751	View CHECK enhancements		
F812	Core	Basic flagging	
F813	Extended flagging		
F821	Local table references		
F831	Full cursor update		
F831-01	Updatable scrollable cursors		
F831-02	Updatable ordered cursors		
F841	LIKE_REGEX predicate		
F842	OCCURENCES_REGEX function		

F842	OCCURENCES_REGEX function		
F843	POSITION_REGEX function		
F844	SUBSTRING_REGEX function		
F845	TRANSLATE_REGEX function		
F846	Octet support in regular expression operators		
F847	Nonconstant regular expressions		
F866	FETCH FIRST clause: PERCENT option		
F867	FETCH FIRST clause: WITH TIES option		
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025	Final structured types		
S026	Self-referencing structured types		
S027	Create method by specific method name		
S028	Permutable UDT options list		
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	partially supported
S081	Enhanced object support	Subtables	
S091	Basic array support	partially supported	
S091-01	Arrays of built-in data types		
S091-02	Arrays of distinct types		
S091-03	Array expressions		
S094	Arrays of reference types		

S097	Array element assignment	
S151	Basic object support	Type predicate
S161	Enhanced object support	Subtype treatment
S162	Subtype treatment for references	
S202	SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators
S232	Array locators	
S233	Multiset locators	
S241	Transform functions	
S242	Alter transform statement	
S251	User-defined orderings	
S261	Specific type method	
S271	Basic multiset support	
S272	Multisets of user-defined types	
S274	Multisets of reference types	
S275	Advanced multiset support	
S281	Nested collection types	
S291	Unique constraint on entire row	
S301	Enhanced UNNEST	
S401	Distinct types based on array types	
S402	Distinct types based on distinct types	
S403	ARRAY_MAX_CARDINALITY	
S404	TRIM_ARRAY	
T011	Timestamp in Information Schema	
T021	BINARY and VARBINARY data types	
T022	Advanced support for BINARY and VARBINARY data types	
T023	Compound binary literal	
T024	Spaces in binary literals	
T041	Basic object support	Basic LOB data type support

		support
T041-01	Basic object support	BLOB data type
T041-02	Basic object support	CLOB data type
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types
T041-04	Basic object support	Concatenation of LOB data types
T041-05	Basic object support	LOB locator: non-holdable
T042	Extended LOB data type support	
T043	Multiplier T	
T044	Multiplier P	
T051	Row types	
T052	MAX and MIN for row types	
T053	Explicit aliases for all-fields reference	
T061	UCS support	
T101	Enhanced nullability determination	
T111	Updatable joins, unions, and columns	
T174	Identity columns	
T175	Generated columns	
T176	Sequence generator support	
T177	Sequence generator support: simple restart option	
T178	Identity columns: simple restart option	
T180	System-versioned tables	
T181	Application-time period tables	
T211	Active database, Enhanced integrity management	Basic trigger capability
T211-	Active database, Enhanced	Support for run-time rules

06	integrity management	triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251	SET TRANSACTION statement: LOCAL option		
T261	Chained transactions		
T272	Enhanced savepoint management		
T285	Enhanced derived column names		
T301	Functional dependencies	partially supported	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T322	PSM	Declared data type attributes	
T324	Explicit security for SQL routines		
T325	Qualified SQL parameter references		
T326	Table functions		
T332	Extended roles	mostly supported	
T431	OLAP	Extended grouping capabilities	
T432	Nested and concatenated GROUPING SETS		
T433	Multiargument GROUPING function		
T434	GROUP BY DISTINCT		
T471	Result sets return value		
T472	DESCRIBE CURSOR		

T495	Combined data change and retrieval	different syntax	
T502	Period predicates		
T511	Transaction counts		
T521	Named arguments in CALL statement		
T522	Default values for IN parameters of SQL-invoked procedures		
T541	Updatable table references		
T561	Holdable locators		
T571	Array-returning external SQL-invoked functions		
T572	Multiset-returning external SQL-invoked functions		
T601	Local cursor references		
T611	OLAP	Elementary OLAP operations	most forms supported
T612	Advanced OLAP operations	some forms supported	
T613	Sampling		
T616	Null treatment option for LEAD and LAG functions		
T618	NTH_VALUE function	function exists, but some options missing	
T619	Nested window functions		
T620	WINDOW clause: GROUPS option		
T641	Multiple column assignment	only some syntax variants supported	
T652	SQL-dynamic statements in SQL routines		
T653	SQL-schema statements in external routines		
T654	SQL-dynamic statements in external routines		
M001	Datalinks		
M002	Datalinks via SQL/CLI		
M003	Datalinks via Embedded SQL		

M005	Foreign schema support	
M006	GetSQLString routine	
M007	TransmitRequest	
M009	GetOpts and GetStatistics routines	
M010	Foreign data wrapper support	different API
M011	Datalinks via Ada	
M012	Datalinks via C	
M013	Datalinks via COBOL	
M014	Datalinks via Fortran	
M015	Datalinks via M	
M016	Datalinks via Pascal	
M017	Datalinks via PL/I	
M018	Foreign data wrapper interface routines in Ada	
M019	Foreign data wrapper interface routines in C	different API
M020	Foreign data wrapper interface routines in COBOL	
M021	Foreign data wrapper interface routines in Fortran	
M022	Foreign data wrapper interface routines in MUMPS	
M023	Foreign data wrapper interface routines in Pascal	
M024	Foreign data wrapper interface routines in PL/I	
M030	SQL-server foreign data support	
M031	Foreign data wrapper general routines	
X012	Multisets of XML type	
X013	Distinct types of XML type	
X014	Attributes of XML type	
X015	Fields of XML type	
X025	XMLCast	

X030	XMLDocument
X038	XMLText
X065	XMLParse: BLOB input and CONTENT option
X066	XMLParse: BLOB input and DOCUMENT option
X068	XMLSerialize: BOM
X069	XMLSerialize: INDENT
X073	XMLSerialize: BLOB serialization and CONTENT option
X074	XMLSerialize: BLOB serialization and DOCUMENT option
X075	XMLSerialize: BLOB serialization
X076	XMLSerialize: VERSION
X077	XMLSerialize: explicit ENCODING option
X078	XMLSerialize: explicit XML declaration
X080	Namespaces in XML publishing
X081	Query-level XML namespace declarations
X082	XML namespace declarations in DML
X083	XML namespace declarations in DDL
X084	XML namespace declarations in compound statements
X085	Predefined namespace prefixes
X086	XML namespace declarations in XMLTable
X091	XML content predicate
X096	XMLExists
X100	Host language support for XML: CONTENT option
X101	Host language support for XML: DOCUMENT option
	Host language support for XML:

	VARCHAR mapping
X111	Host language support for XML: CLOB mapping
X112	Host language support for XML: BLOB mapping
X113	Host language support for XML: STRIP WHITESPACE option
X114	Host language support for XML: PRESERVE WHITESPACE option
X131	Query-level XMLBINARY clause
X132	XMLBINARY clause in DML
X133	XMLBINARY clause in DDL
X134	XMLBINARY clause in compound statements
X135	XMLBINARY clause in subqueries
X141	IS VALID predicate: data-driven case
X142	IS VALID predicate: ACCORDING TO clause
X143	IS VALID predicate: ELEMENT clause
X144	IS VALID predicate: schema location
X145	IS VALID predicate outside check constraints
X151	IS VALID predicate with DOCUMENT option
X152	IS VALID predicate with CONTENT option
X153	IS VALID predicate with SEQUENCE option
X155	IS VALID predicate: NAMESPACE without ELEMENT clause
X157	IS VALID predicate: NO NAMESPACE with ELEMENT clause
X160	Basic Information Schema for registered XML Schemas
	Advanced Information Schema for

X161	Advanced Information Schema for registered XML Schemas
X170	XML null handling options
X171	NIL ON NO CONTENT option
X181	XML(DOCUMENT(UNTYPED)) type
X182	XML(DOCUMENT(ANY)) type
X190	XML(SEQUENCE) type
X191	XML(DOCUMENT(XMLSCHEMA)) type
X192	XML(CONTENT(XMLSCHEMA)) type
X200	XMLQuery
X201	XMLQuery: RETURNING CONTENT
X202	XMLQuery: RETURNING SEQUENCE
X203	XMLQuery: passing a context item
X204	XMLQuery: initializing an XQuery variable
X205	XMLQuery: EMPTY ON EMPTY option
X206	XMLQuery: NULL ON EMPTY option
X211	XML 1.1 support
X221	XML passing mechanism BY VALUE
X222	XML passing mechanism BY REF
X231	XML(CONTENT(UNTYPED)) type
X232	XML(CONTENT(ANY)) type
X241	RETURNING CONTENT in XML publishing
X242	RETURNING SEQUENCE in XML publishing
X251	Persistent XML values of XML(DOCUMENT(UNTYPED)) type

X252	XML(DOCUMENT(ANY)) type
X253	Persistent XML values of XML(CONTENT(UNTYPED)) type
X254	Persistent XML values of XML(CONTENT(ANY)) type
X255	Persistent XML values of XML(SEQUENCE) type
X256	Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type
X257	Persistent XML values of XML(CONTENT(XMLSCHEMA)) type
X260	XML type: ELEMENT clause
X261	XML type: NAMESPACE without ELEMENT clause
X263	XML type: NO NAMESPACE with ELEMENT clause
X264	XML type: schema location
X271	XMLValidate: data-driven case
X272	XMLValidate: ACCORDING TO clause
X273	XMLValidate: ELEMENT clause
X274	XMLValidate: schema location
X281	XMLValidate: with DOCUMENT option
X282	XMLValidate with CONTENT option
X283	XMLValidate with SEQUENCE option
X284	XMLValidate NAMESPACE without ELEMENT clause
X286	XMLValidate: NO NAMESPACE with ELEMENT clause
X300	XMLTable
X301	XMLTable: derived column list option
X302	XMLTable: ordinality column option

X303	XMLTable: column default option
X304	XMLTable: passing a context item
X305	XMLTable: initializing an XQuery variable

Appendix E. 版本说明

Table of Contents

版本说明包含每个PostgreSQL版本中重大变化， 在上面列出的主要特性和迁移问题。 版本说明不包含只影响少数用户或者改变是内部的不是用户可见的那些变化。 比如，在几乎每一个版本中都会改进优化器。但是这种改进 往往通过用户作为简单快速查询被观察。

每个版本变化的完整列表可以通过查看每个版本的Git日志 获得。 `pgsql-committers` email list 也记录所有源代码变化。 还有网络 接口显示了特定文件的变化。

每项目附近出现的名称代表了该项目的主要开发人员。当然 所有的变化包含社区讨论和补丁检查，因此每个项目真正的是社区的努力。

E.1. 版本 9.3.1

> 发布日期: 2013-10-10

这个版本包含各种自9.3.0以来的修复。关于9.3主版本的新特性信息， 请查看Section E.2。

E.1.1. 迁移到版本 9.3.1

运行9.3.X的用户不需要转储/恢复。

E.1.2. 修改列表

- 用JSON功能更新hstore扩展 (Andrew Dunstan)

  在9.3.2之前安装了hstore的用户必须执行：

  ```
  ALTER EXTENSION hstore UPDATE;
  ```

  添加两个新的JSON函数和一个计算。

- 修复创建范围索引时的内存泄露 (Heikki Linnakangas)
- 序列化快照修复 (Kevin Grittner, Heikki Linnakangas)
- 修复libpq SSL死锁错误 (Stephen Frost)
- 修复pg_receivexlog中的时间线处理错误 (Heikki Linnakangas, Andrew Gierth)
- 阻止`CREATE FUNCTION`检查`SET`变量，除非启用了函数体检查 (Tom Lane)
- 删除清理没有索引的表期间罕见的不正确的警告 (Heikki Linnakangas)

E.2. 版本 9.3

> 发布日期: 2013-09-09

E.2.1. 概述

PostgreSQL 9.3中主要的改进包括：

- 添加物化视图
- 让简单的视图可自动更新
- 为 `JSON` 数据类型添加许多特性，包括操作符和函数，从 `JSON` 值中提取元素
- 为 `FROM` 子句子查询和函数调用实现了SQL标准的 `LATERAL` 选项
- 允许外部数据封装器 支持在外部表上书写（插入/更新/删除）
- 添加一个Postgres外部数据封装器，允许访问其他Postgres服务器
- 添加对事件触发器的支持
- 添加选择能力到checksum 数据页和报告损坏
- 阻止非键字段行更新阻塞外键检查
- 大大的减少System V 共享内存的需求

下面的章节是对以上条例更加详细的解释。

E.2.2. 迁移到版本 9.3

对于那些想要从任何以前的版本中迁移数据的人来说，需要使用 pg_dumpall或 pg_upgrade 转储/恢复。

版本9.3包含一些可能影响与以前版本兼容性的修改。观察下列的不兼容性：

E.2.2.1. 服务器设置

- 重命名 `replication_timeout` 为 `wal_sender_timeout` (Amit Kapila)

  这些设置控制WAL发送超时。

- 要求超级用户权限设置 `commit_delay`， 因为它现在可以潜在的延迟其他会话 (Simon Riggs)
- 允许内存中排序使用它们分配的所有内存 (Jeff Janes)

  基于以前的行为设置 `work_mem` 的用户可能需要重新访问该设置。

E.2.2.2. 其他

- 如果一个元组在被更新或删除之前早已被一个 `BEFORE` 触发器更新或删除了， 则抛出一个错误 (Kevin Grittner)

  以前，最初的更新默默的跳过了，导致逻辑上的不一致性，因为触发器可能基于计划更新传播数据到其他地方。 现在抛出了一个错误，阻止不一致的结果被提交。如果这个改变影响了你的应用， 那么最好的解决方法通常是移动数据传播动作到一个 `AFTER` 触发器。

  如果一个查询中调用了一个不稳定的函数修改了稍后该查询本身修改的行，也会抛出这个错误。 这样的情况以前导致默默的跳过更新。

- 修改多行字段 `ON UPDATE SET NULL/SET DEFAULT` 外键动作，影响该约束的所有字段，不只是那些在 `UPDATE` 中修改了的字段 (Tom Lane)

  以前，我们将只设置这些对应于 `UPDATE` 修改过的被引用字段的引用字段。 这是SQL-92要求的，但是最近的SQL标准版本指定了新的行为。

- 如果 `search_path` 改变了， 则强制缓存的计划重新规划 (Tom Lane)

  以前，如果查询用一个新的 `search_path` 设置重新执行， 那么早已在当前的会话中生成的缓存的计划是不会重做的，导致意外的行为。

- 修复 `to_number()`， 正确的处理一个句点用作分隔符 (Tom Lane)

  以前，一个句点被看做是一个小数点，即使本地说它不是并且使用 `D` 格式代码声明本地特定的小数点的使用。如果也使用了 `FM` 格式， 那么这会导致错误的答案。

- 修复 `STRICT` 非设置返回函数，让在它们的参数中的设置返回函数正确的返回空行 (Tom Lane)

  传递到严格函数的空值应该导致一个空的输出，但是相反的，输出行彻底被压制。

- 在一个连续的流中存储WAL， 而不是每4GB就跳过最后16MB段 (Heikki Linnakangas)

  以前，由于这个跳过，以 `FF` 结束名字的WAL文件是不使用的。 如果你有WAL备份或恢复脚本考虑到这种行为，将需要调整它们。

- 在 `pg_constraint.confmatchtype` 中， 存储缺省的外键匹配类型(non- `FULL` , non- `PARTIAL`) 为 `s` ， 代表"简单的" (Tom Lane)

以前这种情况通过 `u` 表示，代表"未指定"。

E.2.3. 修改列表

下面你将找到PostgreSQL 9.3和以前的主版本之间详细的变化。

E.2.3.1. 服务器

E.2.3.1.1. 锁定

- 阻止非键字段行更新阻塞外键检查 (Álvaro Herrera, Noah Misch, Andres Freund, Alexander Shulgin, Marti Raudsepp, Alexander Shulgin)

  这个修改提高了并发性，并减少了更新的表包含在外键约束中时死锁的可能性。`UPDATE` 并不改变任何现在在行上采用新的 `NO KEY UPDATE` 锁模式的外键中引用的字段，虽然外键约束使用新的 `KEY SHARE` 锁模式， 这与 `NO KEY UPDATE` 并不冲突。所以这里没有阻塞，除非修改了外键字段。

- 添加配置变量 `lock_timeout`， 允许限制一个会话将等待多长时间去请求任何一个锁 (Zoltán Böszörményi)

E.2.3.1.2. 索引

- 为范围数据类型添加SP-GiST支持 (Alexander Korotkov)
- 允许不记录GiST索引 (Jeevan Chalke)
- 提高GiST索引插入的性能，通过有多个同样好的选择时随机化选择页面来实现 (Heikki Linnakangas)
- 改善哈希索引操作的并发性 (Robert Haas)

E.2.3.1.3. 优化程序

- 为范围类型收集并使用上界、下界和范围长度的直方图 (Alexander Korotkov)
- 为索引访问改善优化器的成本估算 (Tom Lane)
- 为通过散列聚集实现 `DISTINCT` 改善优化器的哈希表尺寸估计 (Tom Lane)
- 抑制非操作符结果并限制规划节点 (Kyotaro Horiguchi, Amit Kapila, Tom Lane)
- 当规划器只关心总的开销时，通过没有保持规划在基本的便宜启动成本上减少规划器开销 (Tom Lane)

E.2.3.1.4. 一般性能

- 添加 `COPY FREEZE` 选项， 避免稍后标记元组为冻结的开销 (Simon Riggs, Jeff Davis)
- 改善 `NUMERIC` 计算的性能 (Kyotaro Horiguchi)
- 改善等待 `commit_delay` 的会话的同步 (Peter Geoghegan)

  这大大的提高了 `commit_delay` 的有效性。
- 通过不要截断还未接触任何临时表的事务中的临时表，提高 `CREATE TEMPORARY TABLE ... ON COMMIT DELETE ROWS` 选项的性能 (Heikki Linnakangas)
- 让清理在删除过期的元组之后重新检查可见性 (Pavan Deolasee)

  这提高了页面被标记为所有可见的可能性。
- 添加每资源所有者锁缓存 (Jeff Janes)

  这加速了持有多个锁的多语句事务中语句完成时的锁统计； 这对于pg_dump尤其有用。
- 避免在一个创建新的关系的事务提交时扫描整个关系缓存 (Jeff Janes)

  这加速了在连续的小事务中创建许多表的会话，比如运行一个pg_restore 。
- 提高删除许多关系的事务的性能 (Tomas Vondra)

E.2.3.1.5. 监视

- 添加可选的能力到checksum 数据页并报告损坏 (Simon Riggs, Jeff Davis, Greg Smith, Ants Aasma)

  checksum选项可以在initdb期间设置。
- 分离统计收集器的数据文件到独立的全局和每数据库文件 (Tomas Vondra)

  这减少了统计数据追踪的I/O需求。
- 修复统计收集器，以便在系统时钟倒退的情况下正确的操作 (Tom Lane)

  以前，统计收集器会停止，直到时间再次到达最后记录的时间。
- 当我们想要在这里停止记录时，发出一个告知性的消息到主进程标准错误 (Tom Lane)

  这会帮助用户减少在主进程启动期间只记录标准错误的常见配置中，在哪里查看日志输出的混淆。

E.2.3.1.6. 认证

- 当发生认证失败时，记录相关的 `pg_hba.conf` 行， 简化意外失败的调试 (Magnus Hagander)
- 改进LDAP错误报告和文档 (Peter Eisentraut)

- 在URL格式中添加对指定LDAP认证参数的支持，每RFC 4516 (Peter Eisentraut)
- 修改 `ssl_ciphers` 参数， 以 `DEFAULT` 启动，而不是以 `ALL` 启动， 然后删除不安全的密码 (Magnus Hagander)

  这会产生一个更合适的SSL密码设置。
- 分析并加载 `pg_ident.conf` 一次， 而不是在每个连接中 (Amit Kapila)

  这类似于 `pg_hba.conf` 的处理。

E.2.3.1.7. 服务器设置

- 大大的减少了System V 共享内存的需求 (Robert Haas)

  在类Unix的系统上， `mmap()` 现在用于大多数PostgreSQL 的共享内存。对于大多数用户， 这将消除任何为共享内存调整内核参数的需要。
- 允许主进程监听多个Unix域套接字 (Honza Horák)

  配置参数 `unix_socket_directory` 被 `unix_socket_directories` 取代， 它接受一个路径列表。
- 允许配置文件的路径被处理 (Magnus Hagander, Greg Smith, Selena Deckelmann)

  这样一个路径在服务器配置文件中用 `include_dir` 指定。
- 为 `shared_buffers` 增加最大initdb配置值到128MB (Robert Haas)

  这是initdb打算在 `postgresql.conf` 中设置的最大值；以前的最大值是32MB。
- 在主进程退出时， 删除外部PID文件， 如果有 (Peter Eisentraut)

E.2.3.2. 复制和恢复

- 允许流复制备用遵循时间线切换 (Heikki Linnakangas)

  这允许流备用服务器从一个新晋升为主要地位的伺服接受WAL数据。以前， 其他备用会请求重新同步以开始跟随新的主机。
- 添加SQL函数 `pg_is_in_backup()` 和 `pg_backup_start_time()` (Gilles Darold)

  这些函数报告基础备份的状态。
- 提高 `synchronous_commit` 禁用时流日志切换的性能 (Andres Freund)
- 允许更快的晋升流备用为主服务器 (Simon Riggs, Kyotaro Horiguchi)
- 添加最后检查点的重做位置到 pg_controldata 的输出 (Fujii Masao)

这个信息对于确定恢复需要哪个WAL文件是有帮助的。

- 允许像pg_receivexlog 这样的工具用不同的架构在计算机上运行 (Heikki Linnakangas)

  WAL文件仍然只能以原先相同的架构在服务器上重放；但是他们现在可以以任意架构在机器上传输和存储， 因为流复制协议现在是与机器无关的。

- 让pg_basebackup `--write-recovery-conf` 输出一个最小的 `recovery.conf` 文件 (Zoltán Böszörményi, Magnus Hagander)

  这简化了设置一个备用服务器。

- 允许pg_receivexlog 和pg_basebackup `--xlog-method` 处理流时间线切换 (Heikki Linnakangas)

- 添加 `wal_receiver_timeout` 参数控制WAL接收器的超时 (Amit Kapila)

  这允许更快速的检测连接失败。

- 改变WAL记录格式， 以允许跨页面分离记录头 (Heikki Linnakangas)

  新的格式更紧凑，并且书写更有效。

E.2.3.3. 查询

- 为 `FROM` 子句子查询和函数调用实现SQL标准的 `LATERAL` 选项 (Tom Lane)

  这个特性允许 `FROM` 中的子查询和函数从 `FROM` 子句中的其他表中引用字段。 `LATERAL` 关键字对于函数来说是可选的。

- 添加对输送 `COPY` 和 psql `\copy` 数据到/从一个外部程序的支持 (Etsuro Fujita)

- 允许规则中的一个多行 `VALUES` 子句引用 `OLD` / `NEW` (Tom Lane)

E.2.3.4. 对象操作

- 添加对事件触发器的支持 (Dimitri Fontaine, Robert Haas, Álvaro Herrera)

  这允许DDL命令运行时，以启用事件的语言书写的服务器端的函数被调用。

- 允许外部数据封装器 支持在外部表上写入（插入/更新/删除） (KaiGai Kohei)

- 添加 `CREATE SCHEMA ... IF NOT EXISTS` 子句 (Fabrízio de Royes Mello)

- 让 `REASSIGN OWNED` 也修改共享对象的所有权 (Álvaro Herrera)

- 如果给定的初始值字符串对于事务数据类型来说不是有效的输入， 那么让 `CREATE AGGREGATE` 投诉 (Tom Lane)

- 抑制 `CREATE TABLE` 关于隐式索引和序列创建的消息 (Robert Haas)

  这些消息现在在 `DEBUG1` 冗长模式中出现，所以它们缺省将不显示。

- 当一个不存在的模式在表名中指定时，允许 `DROP TABLE IF EXISTS` 成功 (Bruce Momjian)

  以前，如果该模式不存在，它抛出一个错误。

- 提供带有约束违反细节 的客户端作为单独的字段 (Pavel Stehule)

  这允许客户端检索表、字段、数据类型或约束名错误细节。以前这样的信息必须从错误字符串中提取。 客户端库支持需要访问这些字段。

E.2.3.4.1. `ALTER`

- 在 `ALTER TYPE ... ADD VALUE` 中支持 `IF NOT EXISTS` 选项 (Andrew Dunstan)

  这对于有条件的添加值到枚举类型是有用的。

- 添加 `ALTER ROLE ALL SET` 为所有用户建立设置 (Peter Eisentraut)

  这允许设置应用到所有数据库中的所有用户。 `ALTER DATABASE SET` 早已允许在单个数据库中为所有用户添加设置。 `postgresql.conf` 也有类似作用。

- 为 `ALTER RULE ... RENAME` 添加支持 (Ali Dar)

E.2.3.4.2. `VIEWs`

- 添加物化视图 (Kevin Grittner)

  不像普通视图，基础表是读取每个访问的，物化视图在创建或刷新时创建物理表。 访问物化视图然后从它的物理表中读取。 现在还没有任何增量刷新物化视图或通过基础表访问自动访问它们的便利。

- 让简单视图自动可更新 (Dean Rasheed)

  从一个基础表中引用一些或所有字段的简单视图现在缺省是可更新的。 更复杂的视图可以使用 `INSTEAD OF` 触发器或 `INSTEAD` 规则使其可更新。

- 添加 `CREATE RECURSIVE VIEW` 语法 (Peter Eisentraut)

  内部的，这些翻译为 `CREATE VIEW ... WITH RECURSIVE ...`。

- 改善视图/规则打印代码，以处理引用的表重命名或字段重命名、添加或删除了的情况 (Tom Lane)

  表和字段重命名会产生的情况是：如果我们只是替代新的名称为一个规则或视图的原始文本， 那么结果是有歧义的。这个修改修复了规则转储代码，当需要保存原来的语义时， 插入手动制作的表和字段别名。

E.2.3.5. 数据类型

- 增加大对象的最大尺寸从2GB到4TB (Nozomi Anzai, Yugo Nagata)

  这个修改包括添加64位大对象访问功能，在服务器中和libpq中。

- 允许文本的时区名称，比如， "America/Chicago"，在ISO格式的 `timestamptz` 输入的"T"字段中 (Bruce Momjian)

E.2.3.5.1. `JSON`

- 添加操作符和函数从 `JSON` 值中提取元素 (Andrew Dunstan)
- 允许 `JSON` 值被转换为记录 (Andrew Dunstan)
- 添加函数转换标量、记录和 `hstore` 值为 `JSON` (Andrew Dunstan)

E.2.3.6. 函数

- 添加 `array_remove()` 和 `array_replace()` 函数 (Marco Nenciarini, Gabriele Bartolini)
- 允许 `concat()` 和 `format()` 正确的扩展 `VARIADIC` 标签的参数 (Pavel Stehule)
- 改进 `format()`， 提供字段宽度和左/右对齐选项 (Pavel Stehule)
- 让 `to_char()`, `to_date()`, 和 `to_timestamp()` 正确的处理负的世纪值(BC) (Bruce Momjian)

  以前，该行为不是错误就是与正的/AD处理不一致，比如， 格式标记"IYYY-IW-DY"。

- 让 `to_date()` 和 `to_timestamp()` 在混合ISO和Gregorian周/天命名时返回正确的结果 (Bruce Momjian)
- 在每个 `SELECT` 目标列表项和 `FROM` 项之后， 让 `pg_get_viewdef()` 缺省开始一个新行 (Marko Tiikkaja)

  这减少了视图打印中的线段长度，例如在pg_dump 的输出中。

- 修复 `map_sql_value_to_xml_value()`， 以与基础类型相同的打印方式， 打印域类型的值 (Pavel Stehule)

  对于某些内建类型，比如 `boolean`， 有特殊的格式规则； 这些规则现在也应用到这些类型上的域。

E.2.3.7. 服务器端的语言

E.2.3.7.1. PL/pgSQL服务器端语言

- 允许PL/pgSQL使用带有复合类型表达式的 `RETURN` (Asif Rehman)

  以前，在一个返回复合类型的函数中， `RETURN` 只能引用一个复合类型的变量。

- 允许PL/pgSQL作为单独的字段存取约束违反细节 (Pavel Stehule)

- 允许PL/pgSQL访问 `COPY` 处理的行数 (Pavel Stehule)

  PL/pgSQL函数中执行的 `COPY` 现在更新通过 `GET DIAGNOSTICS x = ROW_COUNT` 恢复的值。

- 允许未保留的关键字用作PL/pgSQL中的标识符 (Tom Lane)

  在PL/pgSQL语法中的某些地方，关键字必须加引号用作标识符，即使它们名义上是未保留的。

E.2.3.7.2. PL/Python服务器端语言

- 添加PL/Python结果对象字符串处理器 (Peter Eisentraut)

  这允许 `plpy.debug(rv)` 输出合理的东西。

- 让PL/Python转化OID值为一个适当的Python数值类型 (Peter Eisentraut)

- 处理和内部SPI错误一样明确发生(用PL/Python的 `RAISE`) 的SPI错误 (Oskari Saarenmaa and Jan Urbanski)

E.2.3.8. 服务器编程接口(SPI)

- 阻止SPI元组表在子事务退出期间泄露 (Tom Lane)

  在任何失败的子事务的结尾，内核SPI代码现在释放任何在该子事务期间创建的SPI元组表。 这避免了对使用SPI代码的需要，保持在错误恢复代码中追踪这样的元组表和手动释放它们。 未能这样做会在PL/pgSQL和可能其他SPI客户端中导致一些事物寿命内存泄露问题。 `SPI_freetuptable()` 现在保护自己免受多次释放请求，所以任何现有代码需要小心清理， 不应该被这个修改破坏。

- 允许SPI函数访问被 `COPY` 处理的行数 (Pavel Stehule)

E.2.3.9. 客户端应用

- 添加命令行工具pg_isready， 检查服务器是否准备好了接受连接 (Phil Sorber)

- 为pg_restore, clusterdb, reindexdb, 和vacuumdb 支持多个 `--table` 参数 (Josh Kupershmidt)

  这类似于pg_dump的 `--table` 选项的工作方式。

- 添加 `--dbname` 选项到 pg_dumpall, pg_basebackup, 和pg_receivexlog， 允许指定一个连接字符串 (Amit Kapila)
- 添加libpq函数 `PQconninfo()` ， 返回连接信息 (Zoltán Böszörményi, Magnus Hagander)

E.2.3.9.1. psql

- 调整函数开销设置，这样psql选项卡实现和模式搜索更有效 (Tom Lane)
- 提高psql的选项卡实现覆盖(Jeff Janes, Dean Rasheed, Peter Eisentraut, Magnus Hagander)
- 允许psql `--single-transaction` 模式在从标准输入中读取时工作 (Fabien Coelho, Robert Haas)

  以前这个选项只在从一个文件中读取时工作。
- 当连接到一个老的服务器时， 删除psql警告 (Peter Eisentraut)

  当连接到一个比psql的主版本更新的服务器时，仍然发出一个警告。

E.2.3.9.1.1. 反斜杠命令

- 添加psql命令 `\watch` ， 重复的执行一个SQL命令 (Will Leinweber)
- 添加psql命令 `\gset` ， 在psql变量中存储查询结果 (Pavel Stehule)
- 添加SSL信息到psql的 `\conninfo` 命令 (Alastair Turner)
- 添加"Security"字段到psql的 `\df+` 输出 (Jon Erdman)
- 允许psql命令 `\l` 接受一个数据库名字模式 (Peter Eisentraut)
- 在psql中，如果没有活动的连接，则不允许 `\connect` 使用缺省 (Bruce Momjian)

  如果服务器崩溃可能会出现这种情况。
- 在用psql的 `\g` `_file_` SQL命令执行失败之后正确的重置状态 (Tom Lane)

  以前，来自随后的SQL命令的输出会意外的继续进行同一个文件。

E.2.3.9.1.2. 输出

- 添加一个 `latex-longtable` 输出格式到psql (Bruce Momjian)

  这个格式允许表跨越多个页面。
- 添加 `border=3` 输出模式到psql `latex` 格式 (Bruce Momjian)
- 在psql的仅元组和扩展输出模式中，不再为零行发出"(No rows)" (Peter Eisentraut)

- 在psql的未对齐、扩展输出模式中，不再为零行输出一个空行 (Peter Eisentraut)

E.2.3.9.2. pg_dump

- 添加pg_dump `--jobs` 选项并行转储表 (Joachim Wieland)
- 让pg_dump输出函数有一个更加可预见的顺序 (Joel Jacobson)
- 修复pg_dump发出的tar文件，与POSIX一致 (Brian Weaver, Tom Lane)
- 添加 `--dbname` 选项到pg_dump， 与其他客户端命令一致 (Heikki Linnakangas)

  最后提供的数据库名可能没有标志。

E.2.3.9.3. initdb

- 让initdb同步最近创建的数据目录 (Jeff Davis)

  这确保了在initdb之后很快系统崩溃情况下的数据完整性。 可以使用 `--nosync` 禁用此功能。

- 添加initdb `--sync-only` 选项同步数据目录到持久存储 (Bruce Momjian)

  这是通过pg_upgrade使用的。

- 在文件系统的挂载点上放置数据目录时，让initdb发出一个警告 (Bruce Momjian)

E.2.3.10. 源代码

- 添加基础设施以允许插件后端工作进程 (Álvaro Herrera)
- 创建一个集中的超时API (Zoltán Böszörményi)
- 创建libpgcommon并删除这里的 `pg_malloc()` 和其他函数 (Álvaro Herrera, Andres Freund)

  这允许libpgport只用于可移植性相关的代码。

- 为嵌入在较大结构内的列表连接添加支持 (Andres Freund)
- 为所有信号使用 `SA_RESTART`， 包括 `SIGALRM` (Tom Lane)
- 确保在翻译 `errcontext()` 消息时使用了正确的文本域 (Heikki Linnakangas)
- 标准化命名客户端侧的内存分配函数 (Tom Lane)
- 如果某些编译时常量条件不符合，那么为"静态断言" 提供支持将会在编译时失败 (Andres Freund, Tom Lane)
- 在客户端侧代码中支持 `Assert()` (Andrew Dunstan)

- 添加修饰以通知C编译器一些 `ereport()` 和 `elog()` 调用不返回 (Peter Eisentraut, Andres Freund, Tom Lane, Heikki Linnakangas)
- 允许选项通过 `PG_REGRESS_DIFF_OPTS` 传递到回归测试输出比较工具中 (Peter Eisentraut)
- 为 `CREATE INDEX CONCURRENTLY` 添加隔离测试 (Abhijit Menon-Sen)
- 删除类型定义 `int2` / `int4` ， 因为他们更好的表现为 `int16` / `int32` (Peter Eisentraut)
- 修复Mac OS X上的install-strip (Peter Eisentraut)
- 删除configure标志 `--disable-shared` ， 因为不再支持它了 (Bruce Momjian)
- 在Perl中重写pgindent (Andrew Dunstan)
- 提供Emacs宏设置Perl格式匹配PostgreSQL的perltidy设置 (Peter Eisentraut)
- 运行工具检查关键字列表，查看何时后端语法发生了改变 (Tom Lane)
- 改变 `UESCAPE` lex的方式，大大的减少了词法分析程序表的大小 (Heikki Linnakangas)
- 集中flex和bison make规则 (Peter Eisentraut)

  这对于pgxs作者是有用的。
- 修改许多内部后端函数，返回对象 `OID` 而不是空 (Dimitri Fontaine)

  这对于事件触发器是有用的。
- 为事务回调构造pre-commit/pre-prepare/pre-subcommit事件 (Tom Lane)

  使用事务回调的可加载模块可能需要修改，以处理这些新的事件类型。
- 添加函数 `pg_identify_object()` ， 生产一个机器可读的数据库对象的描述 (Álvaro Herrera)
- 添加在 `ALTER` 对象之后的服务器挂钩 (KaiGai Kohei)
- 实现一个通用二进制堆并将它用于合并附加操作 (Abhijit Menon-Sen)
- 提供一个工具在更新 `src/timezone/data` 文件时，帮助检测时区缩写改变 (Tom Lane)
- 为libpq和ecpg库添加pkg-config支持 (Peter Eisentraut)
- 删除 `src/tool/backend` ，因为该内容在PostgreSQL wiki上 (Bruce Momjian)
- 分离WAL读取作为一个独立的设施 (Heikki Linnakangas, Andres Freund)
- 使用64位整数表示WAL位置(`XLogRecPtr`)， 替代两个32位的整数 (Heikki Linnakangas)

  通常，需要读取WAL格式的工具将需要调整。

- 允许PL/Python支持平台特定的包含路径 (Peter Eisentraut)
- 允许OS X上的PL/Python 建立Python的定制版本 (Peter Eisentraut)

E.2.3.11. 附加的模块

- 添加一个Postgres外部数据封装器 贡献模块，以允许访问其他Postgres服务器 (Shigeru Hanada)

  这个外部数据封装器支持写。

- 添加pg_xlogdump贡献程序 (Andres Freund)
- 添加对在pg_trgm中索引正则表达式搜索的支持 (Alexander Korotkov)
- 改善pg_trgm对多字节字符的处理 (Tom Lane)

  在一个没有wcstombs()或towlower()库函数的平台上，这会导致pg_trgm 索引内容中非ASCII数据的不兼容的改变。在这样的情况下， `REINDEX` 这些索引以确保正确的搜索结果。

- 添加一个pgstattuple函数，报告GIN 索引等待插入的列表的大小 (Fujii Masao)
- 让oid2name, pgbench, 和vacuumlo设置 `fallback_application_name` (Amit Kapila)
- 改善pg_test_timing的输出 (Bruce Momjian)
- 改善pg_test_fsync 的输出 (Peter Geoghegan)
- 创建一个专用的外部数据封装器，带有它自己的选项验证器函数， dblink (Shigeru Hanada)

  当使用这个FDW定义一个dblink连接的目标时，取代使用连接选项的硬连线列表， 咨询底层的libpq库查看支持哪个连接选项。

E.2.3.11.1. pg_upgrade

- 允许pg_upgrade并行转储和恢复 (Bruce Momjian, Andrew Dunstan)

  这允许数据库的并行模式转储/恢复，也允许每个表空间并行拷贝/连接数据文件。 使用 `--jobs` 选项指定并行的级别。

- 让pg_upgrade在当前目录中创建Unix域套接字 (Bruce Momjian, Tom Lane)

  这减少了在升级期间某个人意外连接的可能性。

- 让pg_upgrade `--check` 模式正确的检测非缺省套接字目录的位置 (Bruce Momjian, Tom Lane)

- 为拥有许多表的数据库提高pg_upgrade的性能 (Bruce Momjian)
- 通过显示执行的命令改进pg_upgrade的日志 (Álvaro Herrera)
- 改进拷贝/连接期间pg_upgrade的状态显示 (Bruce Momjian)

E.2.3.11.2. pgbench

- 添加 `--foreign-keys` 选项到pgbench (Jeff Janes)

  这添加了外键约束到pgbench创建的标准表，用于外键性能测试。

- 允许pgbench集合性能统计和每 `--aggregate-interval` 秒产生输出 (Tomas Vondra)
- 添加pgbench `--sampling-rate` 选项， 控制事务日志的百分比 (Tomas Vondra)
- 减少并改进pgbench的初始化模式的状态消息输出 (Robert Haas, Peter Eisentraut)
- 添加pgbench `-q` 模式，每5秒打印一行输出 (Tomas Vondra)
- 在初始化期间输出pgbench经过和预估的剩余时间 (Tomas Vondra)
- 允许pgbench使用更大的比例系数，当请求的比例系数超过20000时， 改变相关的字段从 `integer` 到 `bigint` (Greg Smith)

E.2.3.12. 文档

- 允许创建EPUB格式文档 (Peter Eisentraut)
- 更新FreeBSD内核配置文档 (Brad Davis)
- 改进 `WINDOW` 函数文档 (Bruce Momjian, Florian Pflug)
- 添加使用说明， 在Mac OS X上建立文档工具链 (Peter Eisentraut)
- 改进 `commit_delay` 文档 (Peter Geoghegan)

E.3. 版本9.2.5

> 发布日期: 2013-10-10

该版本包含来自9.2.4的各种修复。关于9.2主要版本新功能的信息，请参阅Section E.8。

E.3.1. 迁移到版本9.2.5

为了运行9.2.X不需要转储/恢复。

同时，如果你是从早于9.2.2的版本上更新，参阅9.2.2发布说明。

E.3.2. 变化

- 防止多字节编码中非ASCII非双引号标识符的小写转换(Andrew Dunstan)

  以前的操作是错误的而且混乱的。

- 当创建范围索引时修复内存泄露。(Heikki Linnakangas)
- 当 `wal_level = hot_standby` 的时候，修复后端写进程中检查点内存泄露。(Naoya Anzai)
- 修复通过 `lo_open()` 故障产生的内存泄露。(Heikki Linnakangas)
- 当 `work_mem` 正使用大于24GB的内存时，那么修复内存过量使用错误。(Stephen Frost)
- 可串行化快照修复(Kevin Grittner, Heikki Linnakangas)
- 修复libpq SSL死锁错误(Stephen Frost)
- 修复线程libpq应用中可能的SSL网络堆变化 (Nick Phillips, Stephen Frost)
- 当在通用和自定义计划之间选择时，提高计划成本估计(Tom Lane)

  当计划成本高时，那么该变化将有利于通用计划。

- 正确计算估计布尔列包含许多NULL值的行(Andrew Gierth)

  当估计计划成本时，先前的测试像 `col IS NOT TRUE` 和 `col IS NOT FALSE` 没有合理的NULL值因素。

- 修复 `UNION ALL` 并且继承查询以正确重新检查参数化路径(Tom Lane)

  修复不理想的查询规划潜在地被选择的情况。

- 阻止叠加 `WHERE` 子句到不安全的 `UNION/INTERSECT` 子查询中(Tom Lane)

以前这样叠加可能产生错误。

- 修复通过不恰当地处理日期类型修饰符产生的罕见的 `GROUP BY` 查询错误(Tom Lane)
- 修复有删除列的外表的pg_dump (Andrew Dunstan)

  先前这种情况可能导致pg_upgrade错误。

- 重新安排相关扩展规则的pg_dump处理和事件触发(Joe Conway)
- 如果通过 `pg_dump -t` 或者 `-n` 指定，那么强制扩展表转储(Joe Conway)
- 允许转储编码更好地处理基本表上已删除的列(Tom Lane)
- 使用显示正确格式名的目录归档修复 `pg_restore -l` (Fujii Masao)
- 正确记录使用 `UNIQUE` 和 `PRIMARY KEY` 语法创建的 索引注释(Andres Freund)

  这将修复并行pg_restore故障。

- 造成 `pg_basebackup -x` 使用空xlog目录抛出错误而不是崩溃的原因 (Magnus Hagander, Haruka Takatsuka)
- 清理切换之前合理保证WAL文件传输(Fujii Masao)

  以前，在备库上所有WAL文件被取代之前可能关闭流复制连接。

- 在恢复期间提高WAL段时间线处理(Heikki Linnakangas)
- 修复 `REINDEX TABLE` 和 `REINDEX DATABASE` 以 恰当的重新生效约束并且标记无效索引为有效(Noah Misch)

  `REINDEX INDEX` 一直正常工作。

- 在插入SP-GiST索引期间避免死锁(Teodor Sigaev)
- 在并发 `CREATE INDEX CONCURRENTLY` 操作期间修复可能死锁(Tom Lane)
- 修复GiST索引查找崩溃(Tom Lane)
- 修复 `regexp_matches()` 处理零长度匹配(Jeevan Chalke)

  先前，零长度匹配像'^'可以返回很多匹配。

- 修复过于复杂的正则表达式的错误(Heikki Linnakangas)
- 为反向引用结合非贪婪量词修复正则表达式匹配错误(Jeevan Chalke)
- 避免 `CREATE FUNCTION` 检查 `SET` 变量除非启动函数体检查(Tom Lane)
- 允许 `ALTER DEFAULT PRIVILEGES` 在模式上操作不需要CREATE权限(Tom Lane)

- 放宽用于查询中关键字的限制(Tom Lane)

  特别地，放宽角色名称，语言名字，`EXPLAIN`和`COPY`选项，以及`SET`值的关键字限制。这允许`COPY ... (FORMAT BINARY)`事先`BINARY`需要单引号。

- 在`COPY`失败期间打印合适行数 (Heikki Linnakangas)

- 修复`pgp_pub_decrypt()`因此为带有密码的秘钥工作(Marko Kreen)

- pg_upgrade使用`>pg_dump --quote-all-identifiers`避免在版本之间关键字改变的问题(Tom Lane)

- 在清理无索引表中删除少有的不正确的警告(Heikki Linnakangas)

- 在已取消文件截断请求后提高分析统计生成(Kevin Grittner)

- 当在预备查询中执行事务控制命令(比如`ROLLBACK`)时，避免可能的失败(Tom Lane)

- 允许在所有平台上无穷大的各种拼写(Tom Lane)

  支持无穷大的值是"inf", "+inf", "-inf", "infinity", "+infinity"和"-infinity"。

- 当关闭`track_activities`的时候，避免不必要报告(Tom Lane)

- 扩展记录和数组比较行的能力(Rafal Rzepecki,Tom Lane)

- 当psql的`PSQLRC`变量包含一个波浪号防止崩溃(Bruce Momjian)

- 添加spinlock支持ARM64 (Mark Salter)

- 为了Israel, Morocco, Palestine,Paraguay中DST变化规律的DST变化规律更新时间区域数据文件到tzdata版本2013d。 同时为Macquarie Island修正历史区域数据(Tom Lane)

E.4. 版本9.2.4

> 发布日期: 2013-04-04

该版本包含了9.2.3的各种修复。关于9.2主要版本的新功能信息，请参阅Section E.8。

E.4.1. 迁移到版本9.2.4

为了运行9.2.X不需要转储/恢复。

然而，该版本修正了GiST索引管理的一些错误。在安装这个更新之后， `REINDEX` 任何GiST索引满足一个或更多个下面描述的条件是明智的。

同时，如果你正从9.2.2更早版本中更新，参阅9.2.2的发布说明。

E.4.2. 变化

- 修复服务器命令行开关安全解析(Mitsumasa Kondo, Kyotaro Horiguchi)

  包含以" `-` "开头的数据库名字的连接请求可以用来损坏或者破坏 服务器的数据目录文件，即使最终拒绝该请求。(CVE-2013-1899)

- 在每个postmaster子进程中重置OpenSSL随机状态(Marko Kreen)

  通过 `contrib/pgcrypto` 函数产生的随机数 可能对于另外一个猜测的数据库用户相对容易，避免这种情况。 当postmaster使用 `ssl` = `on` 被配置时，该风险是非常显著的， 但是大多数连接不能使用SSL加密。 (CVE-2013-1900)

- 使用REPLICATION权限检查测试当前用户不是认证用户(Noah Misch)

  一个未经授权的数据库用户可以利用这个错误调用 `pg_start_backup()` 或者 `pg_stop_backup()` , 因此可能干扰常规备份的创建。(CVE-2013-1901)

- 当不适合这样执行时，不使用"fuzzy"几何比较修复GiST索引。(Alexander Korotkov)

  核心几何类型使用"fuzzy"等式执行比较，但是 `gist_box_same` 必须执行 精确比较，否则GiST索引使用它可能变得不一致。安装这个更新之后， 用户应该在 `box` , `polygon` , `circle` 或者 `point` 列上 `REINDEX` 任何GiST索引，因为所有这些使用 `gist_box_same` 。

- 修复不正确的范围并集以及为了可变宽度数据类型使用 `contrib/btree_gist` 的GiST索引中惩罚逻辑， 也就是 `text` , `bytea` , `bit` 和 `numeric` 列(Tom Lane)

这些错误可能导致不一致索引，其中一些出现的关键字不会被搜索发现，并且在无用的索引膨胀中， 在安装此更新后建议用户 `REINDEX` 这种索引。

- 修复为多列索引在GiST页中分离代码的错误(Tom Lane)

  这些错误可能导致不一致索引，其中一些出现的关键字不会被搜索发现， 并且在索引中是不必要的无效的搜索。 在安装此更新后建议用户 `REINDEX` 多列GiST索引。

- 修复 `gist_point_consistent` 处理模糊一致性(Alexander Korotkov)

  在 `point` 列GiST索引上的索引扫描可能有时产生不同于顺序扫描的结果， 因为 `gist_point_consistent` 不同意底层操作编码关于是否精确或者模糊的执行比较。

- 在WAL重放中修复缓冲区泄露(Heikki Linnakangas)

  在回放期间这个缺陷可能产生"不正确的本地针数"错误，使得恢复不可能。

- 确保我们在进入归档恢复前执行故障修复，如果数据库没有规则地中断，并且 `recovery.conf` 文件 是存在的(Heikki Linnakangas, Kyotaro Horiguchi, Mitsumasa Kondo)

  这需要确保数据库在一定情况下的一致性， 比如初始化一个来自运行的服务器上的文件系统快照的备用服务器。

- 避免在崩溃恢复期间删除未归档的WAL文件(Heikki Linnakangas, Fujii Masao)

- 修复 `DELETE RETURNING` 中的紊乱情况(Tom Lane)

  在这样的情况下， `DELETE RETURNING` 试图从当前进程不再有任何针的 共享缓冲区中抓取数据。 如果一些其他进程同时改变缓冲区， 这将导致垃圾 `RETURNING` 输出，甚至崩溃。

- 修复规则表达式编译中的无限循环风险(Tom Lane, Don Porter)

- 修复规则表达式编译中潜在的空指针引用(Tom Lane)

- 合适的地方修复 `to_char()` 只使用ASCII大小写折叠规则(Tom Lane)

  这种修复一些区域独立化的模板模式的不当行为，但是 在Turkish区域中胡乱操作" `I` "和" `i` "。

- 修复时间戳 `1999-12-31 24:00:00` 不必要的拒绝(Tom Lane)

- 修复SQL语言函数可以安全的用于支持范围类型的函数(Tom Lane)

- 当一个事务执行 `UNLISTEN` 然后 `LISTEN` 时，修复逻辑错误(Tom Lane)

  该会话根本不监听通知事件，尽管它确实在这种情况下应该监听。

- 在列被添加到依赖于其他视图的视图中之后修复可能的规划器崩溃(Tom Lane)

- 修复`EXPLAIN (ANALYZE, TIMING OFF)`中的性能问题(Pavel Stehule)
- 删除无效的"picksplit不支持的二次分裂"日志消息(Josh Hansen, Tom Lane)

  该消息似乎被添加到从未写入的期望代码中，并且可能从来不是，因为二次分裂的GiST的缺省处理 实际上相当好。所以停止打扰关于它的最终用户。

- 删除`gist_box_picksplit()`中残留的二次分裂支持(Tom Lane)

  这不仅是二次分裂的实现不如缺省实现，它实际上更糟。 所以删除它并让缺省代码路径处理该情况。

- 修复发送会话的 最后几个事务提交/终止计数到统计收集器的可能错误(Tom Lane)
- 消除在PL/Perl中的`spi_prepare()`函数的内存泄露(Alex Hunsaker, Tom Lane)
- 修复pg_dumpall以处理正确包含"`=`"的 数据库名字(Heikki Linnakangas)
- 当给定一个不正确的连接字符串时，避免pg_dump中崩溃(Heikki Linnakangas)
- 忽略pg_dump和pg_upgrade中的无效索引(Michael Paquier, Bruce Momjian)

  备份无效索引可能导致恢复时间的问题， 比如如果索引创建失败的原因是它试图强制 不满足表的数据的唯一性条件。同时，如果索引创建实际上仍然在进行中， 认为它是一个不受约束的DDL变化似乎是合理的，其中pg_dump 不期望备份。pg_upgrade现在也跳过无效索引而非失败。

- 在pg_basebackup中，当备份表空间时，仅仅包含当前服务器版本的子目录 (Heikki Linnakangas)
- 在pg_basebackup和 pg_receivexlog中添加服务器版本检查， 因此它们与不工作的版本联合失败(Heikki Linnakangas)
- 修复`contrib/dblink`以安全地处理`DateStyle`或者`IntervalStyle`的 不一致设置(Daniel Farina, Tom Lane)

  先前，如果远程服务器有这些参数的不同设置，可能错误地读取模糊日期。 这个修复确保了通过`dblink`查询抓取的日期时间和间隔列将正确的被解释。 注意然而这个不一致的设置仍然有风险，因为出现在SQL命令中发送到远程服务器的文本值可能 比他们局部的有不同的解释。

- 修复`contrib/pg_trgm`的`similarity()`函数为少于三个的字符串返回零(Tom Lane)

  先前它返回`NaN`由于内部除以零。

- 使用Microsoft Visual Studio 2012启动编译PostgreSQL (Brar Piening, Noah Misch)
- 为了Chile, Haiti, Morocco, Paraguay和一些Russian区域中DST变化规律更 新时间区域数据文件到tzdata版本2013d。 同时为更多地方修正历史区域数据。

同时，为俄罗斯和其他地方的最近变化更新时区缩写文件：`CHOT`，`GET`，`IRKT`，`KGT`，`KRAT`，`MAGT`，`MAWT`，`MSK`，`NOVT`，`OMST`，`TKT`，`VLAT`，`WST`，`YAKT`，`YEKT` 现在遵从他们当前的含义，以及 `VOLT` (Europe/Volgograd)和 `MIST` (Antarctica/Macquarie) 被添加到缺省缩写列表中。

E.5. 版本9.2.3

> 发布日期: 2013-02-07

此版本包含了9.2.2各种修复。关于9.2主要版本的新功能的信息，参阅Section E.8。

E.5.1. 迁移到版本9.2.3

为了运行9.2.X不需要转储/恢复。

然而，如果你是从早于9.2.2的版本上更新，参阅9.2.2发布说明。

E.5.2. 变化

- 防止来自SQL的 `enum_recv` 的执行(Tom Lane)

  该函数被错误声明，允许简单SQL命令导致服务器崩溃。原则上攻击者可以使用它检查服务器内存的内容。 我们该感谢Sumit Soni (通过Secunia SVCRP)报告这个问题。(CVE-2013-0255)

- 当检测WAL回放期间达到一致性数据库状态时，修复多个问题。 (Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- 当不需要实际恢复工作时，修复结束备份点检查(Heikki Linnakangas)

  这个错误可能导致不正确的"在线备份结束之前WAL结束"错误。

- 当截断关系文件时更新最小恢复点(Heikki Linnakangas)

  一旦数据被丢弃，在时间线中的早一点停止恢复不再安全。

- 在改变恢复目标时间线之后修复WAL段重新回收利用(Heikki Linnakangas)

- 正确地恢复来自级联备用服务器上归档的时间线历史文件(Heikki Linnakangas)

- 修复热备份服务器上的锁冲突检测(Andres Freund, Robert Haas)

- 修复热备份模式中错过的取消事件(Noah Misch, Simon Riggs)

  取消冲突的热备查询的需求有时会被错过， 允许这些查询查看不一致数据。

- 用户可以连接之前防止从暂停中恢复暂停功能(Tom Lane)

- 修复SQL语法以允许下标或者来自子SELECT结果的字段选择(Tom Lane)

- 解决繁忙工作负载下的自动清理截断的性能问题(Jan Wieck)

  在表末尾的空白页截断需要排他锁，但是当有冲突的锁请求时，自动清理编码失败（并且释放表锁）。 在负载下，截断不会发生是可能的，导致表膨胀。通过执行部分截断进行修复， 释放锁，然后尝试重新获取锁并且继续。该修复在冲突请求到达后自动清理释放锁之前将大大 减少平均时间。

- 提高 `SPI_execute` 的性能以及相关函数，从而提高了PL/pgSQL的 `EXECUTE` (Heikki Linnakangas, Tom Lane)

  删除一些数据拷贝开销，它被增加到9.2中作为 计划的缓存机制中的修订结果。与9.1相比较 消除了性能回归，也节省了内存，特别是当要执行的查询字符串包含许多 SQL语句的时候。

  另一个好处是，多个语句查询字符串现在 完全连续地被处理， 这是我们在运行解析分析之前完成早期语句的执行以及 在下面一个中进行规划。这消除了长期存在的问题，在影响以后声明操作的DDL中 将表现为预期的。

- 恢复索引用法的pre-9.2成本估算(Tom Lane)

  修正因子的不妥当变化为使用非常大的索引导致不符合要求的高成本估计。

- 修复 `DROP INDEX CONCURRENTLY` 中的间歇崩溃(Tom Lane)

- 修复在 `CREATE/DROP INDEX CONCURRENTLY` 期间的共享内存锁表的潜在败坏 (Tom Lane)

- 在一个元组超过页面大小减去填充因子的情况下修复 `COPY` 的多个元组插入代码(Heikki Linnakangas)

  前面的编码可以进入一个无限循环。

- 当扫描 `pg_tablespace` 的时候防止竞争条件(Stephen Frost, Tom Lane)

  如果有 `pg_tablespace` 项的并发更新， 那么 `CREATE DATABASE` 和 `DROP DATABASE` 可能行为不当。

- 防止 `DROP OWNED` 试图把整个数据库或表空间删除(Álvaro Herrera)

  为安全起见，这些对象的所有权必须被重新分配，而不是删掉。

- 修复 `vacuum_freeze_table_age` 实现中的错误(Andres Freund)

  为了超过 `vacuum_freeze_min_age` 事务已存在的安装中，这个错误阻止使用部分表 扫描自动清理，以致于全表扫描总是相反的。

- 当 `RowExpr` 或者 `XmlExpr` 是解析分析2倍时，防止不当行为(Andres Freund, Tom Lane)

  这个错误可能在上下文中是用户可见的比如 `CREATE TABLE LIKE INCLUDING INDEXES` 。

- 提高哈希表的大小计算中整数溢出的防御(Jeff Davis)
- 解决数据类型权限相关的一些错误(Tom Lane)

  有一些类型的默认权限问题，并且pg_dump失败备份这样的权限。

- 修复服务器崩溃后忽略剩余的临时表的失败(Tom Lane)
- 修复为Windows上大小的原因交替postmaster日志文件的失败(Jeff Janes, Heikki Linnakangas)
- 拒绝 `to_date()` 中超出范围的日期(Hitoshi Harada)
- 修复 `pg_extension_config_dump()` 恰当地处理 扩展更新情况(Tom Lane)

  这个函数将为目标表取代任何现有项， 使其在扩展更新脚本中可用。

- 修复在尽可能简单的表达式中计划时间误差的PL/pgSQL的报告(Tom Lane)

  前面的编码有时导致省略 `CONTEXT` 追踪误差中的第一行。

- 修复函数作为多表触发器的PL/Python的处理(Andres Freund)
- 确保非ASCII提示字符串被转换为Windows上正确的代码页(Alexander Law, Noah Misch)

  这个错误影响psql和一些其他客户端程序。

- 当没有连接到数据库的时候，修复psql的 `\?` 命令中的 可能的崩溃(Meng Qingzhong)
- 当正在运行pg_basebackup的时候，如果删除了关系文件，那么修复 可能的错误(Heikki Linnakangas)
- 当 `pg_basebackup -X fetch` 在备用服务器备份的时候，忍受时间线切换(Heikki Linnakangas)
- 当在热备服务器上运行的时候，使pg_dump排除未记录的表中的数据(Magnus Hagander)

  这将失败，因为在备用服务器上数据是不可用的，所以认为最方便的是自动假设 `--no-unlogged-table-data` 。

- 修复pg_upgrade安全地处理无效索引(Bruce Momjian)
- 修复pg_upgrade的-O/-o选项(Marti Raudsepp)
- 修复libpq的 `PQprintTuples` 中一个字节缓冲区溢出(Xi Wang)

  这个过时的函数不用在PostgreSQL自身的任何地方， 但它可能仍然被客户端代码使用。

- 使得ecpglib正确使用已翻译消息(Chen Huajun)

- 在MSVC上正确安装ecpg_compat和 pgtypes库(Jiang Guiqing)
- 如果它不是通过系统提供的，那么在libecpg中包含 `isinf()` 我们的版本(Jiang Guiqing)
- 为已提供的函数重新安排配置测试， 因此它不会被来自libedit/libreadline的假冒输出愚弄(Christoph Berg)
- 确保Windows随时间编译数增加(Magnus Hagander)
- 当交叉编译Windows时，使得pgxs生成带有正确 `.exe` 后缀的可执行程序(Zoltan Boszormenyi)
- 添加新的时区缩写 `FET` (Tom Lane)

  有一些东欧时区。

E.6. 版本9.2.2

> 发布日期: 2012-12-06

这个版本包含9.2.1的各种修复。有关9.2主要版本的新功能的信息，参阅 Section E.8。

E.6.1. 迁移到版本9.2.2

为了运行9.2.X不需要转储/恢复。

然而，你可能需要执行 `REINDEX` 操作以调整并发建立索引问题， 正如下面描述的第一个changelog项。

同时，如果你正在从版本9.2.0更新，那么请参阅9.2.1的发布说明。

E.6.2. 变化

- 修复与 `CREATE/DROP INDEX CONCURRENTLY` 相关的多个错误(Andres Freund, Tom Lane, Simon Riggs, Pavan Deolasee)

  当在 `CREATE INDEX CONCURRENTLY` 最初阶段添加 `DROP INDEX CONCURRENTLY` 允许不正确索引决定，引入错误；因此通过该命令建立的索引可能败坏。 在应用该更新后推荐使用 `CREATE INDEX CONCURRENTLY` 在9.2.X中重新建立索引。

  另外，当改变索引的 `pg_index` 行状态时，修复 `CREATE/DROP INDEX CONCURRENTLY` 使用合适更新。 这避免竞争条件导致并发会话错过更新目标索引，因此再次导致败坏同时创建索引。

  同时，修复各种其他操作以确保他们忽略了 一个来自失败的 `CREATE INDEX CONCURRENTLY` 命令的无效索引。最重要的是 `VACUUM`， 因为在采用调整动作以修复或者删除无效索引之前在表上可以很容易启动自动清理。

  同时修复 `DROP INDEX CONCURRENTLY` 用来不禁用 插入到目标索引直到所有查询使用它。

  如果取消 `DROP INDEX CONCURRENTLY` :先前编码可以留下未删除的索引，修复不正当操作。

- 为了 `DROP INDEX CONCURRENTLY` 调整谓词锁 (Kevin Grittner)

  先前，在错误的时间处理SSI谓词锁，可能导致与 `DROP` 并行执行的可串行化事务不正确操作。

- 在WAL回放期间修复缓冲区锁定(Tom Lane)

当回放WAL记录影响超过一页时，那么WAL回放编码不能仔细锁定缓冲区。这可能导致热备份查询瞬时看到不一致状态，导致错误结果或者意外失败。

- 修复GIN索引在WAL产生逻辑中的错误(Tom Lane)

  这可能导致索引败坏，如果发生破损页失败。

- 修复SP-GiST索引在WAL回放逻辑错误(Tom Lane)

  这可能导致崩溃后索引败坏，或者是在备用服务器上。

- 在WAL恢复期间修复基础备份位置的不正确检查(Heikki Linnakangas)

  数据库达到一致状态之前该错误允许热备份模式启动。

- 当推动热备服务器正常运行时，正常删除启动进程的虚拟XID锁(Simon Riggs)

  该监督可以防止某种操作的后续执行比如 `CREATE INDEX CONCURRENTLY` 。

- 避免备用模式中假冒的"失序时间线ID"错误(Heikki Linnakangas)
- 在它接收到关机信号之后阻止postmaster发起新的子进程(Tom Lane)

  该错误可能导致比较长的关闭，或者即使从来没有完成没有额外用户操作。

- 当 `log_rotation_age` 超过2^31毫秒时（大约25天），那么修复syslogger进程而不会失败。(Tom Lane)
- 当请求的超时过期的时候，那么修复 `WaitLatch()` 以及时返回(Jeff Janes, Tom Lane)

  与以前的编码，非等待终止中断的稳定流可能会延迟从 `WaitLatch()` 无限期的返回。这已被证明是一个自动清理发射进程中的问题， 可能会导致其他地方的麻烦。

- 当内存不足的时候避免内部哈希表的败坏(Hitoshi Harada)
- 防止已删除表文件描述符在以前事务结束中保持打开(Tom Lane)

  这应该减少长期以来已删除的表继续占用磁盘空间问题。

- 当一个新的子进程无法为它的闭锁创建一个管道的时候，防止数据库端的崩溃和重启(Tom Lane)

  虽然新的进程失败了，没有充分的理由强迫数据库端重新启动， 所以要避免。当内核差不多超出文件描述符的时候，这提高了鲁棒性。

- 避免与加入不平的子查询的规划器崩溃(Tom Lane)
- 修复外连接上的非严格等价从句的规划(Tom Lane)

规划器可以获得来自分句等同于 其他一些的非严格构建的不正确的约束，例如 `WHERE COALESCE(foo, 0) = 0` ， 当 `foo` 来自外连接的空侧。 9.2显示了比以前的版本更多的情况下的这种类型错误， 但是基本的错误已经有很长时间。

- 使用继承树上的索引优化 `MIN` / `MAX` 修复 `SELECT DISTINCT` (Tom Lane)

  该计划在给定的这些因素的结合"未能重新找到MinMaxAggInfo记录"可能失败。

- 确保规划器将隐式和显式转换出于所有目的看作等效的， 除了少数情况下实际上是一个语义差异(Tom Lane)

- 当考虑是否部分索引可用于查询时，包含join子句(Tom Lane)

  严格的join子句可以充分建立一个 `_x_` `IS NOT NULL` 断言，比如。 修复9.2中的规划器回归分析，因为先前版本可能做了可比较推理。

- 当同一索引中有很多可索引join子句时，限制规划器时间的增长(Tom Lane)

- 提高规划器的能力以证明等价类的排除约束(Tom Lane)

- 修复散列子规划中的部分行匹配以正确处理交叉类型例子(Tom Lane)

  这影响到多列 `NOT IN` 子规划，比如 `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` 当例如 `b` 和 `y` 分别为 `int4` 和 `int8` 时。 这个错误导致错误结果或依据所涉及的具体数据类型崩溃。

- 修复btree 标记/恢复函数以处理数组键(Tom Lane)

  这种疏忽可能导致来自 内侧是使用 `_indexed_column_` = ANY(`_array_`)条件的索引扫描的合并联接中的错误结果。

- 为采用更少快照恢复补丁(Tom Lane)

  减少查询执行中采取的快照数的9.2变化 导致一些在以前版本中没有见过的异常行为，因为执行会继续在锁定查询使用的表之前获得的一个快照。 因此，例如， 查询将不保证能够看到前面事务提交的更新 即使该事务有排他锁。我们可能会在未来版本中重新审视它，但同时把它以9.2之前方式放回到原处。

- 当为 `AFTER ROW UPDATE/DELETE` 触发器重新读取旧的元组时，获取缓冲锁(Andres Freund)

  在十分特殊的情况下，这可能会导致传递 不正确的数据到一个触发器 `WHEN` 条件， 或对一个外键执行触发器重新检测逻辑。 这可能导致崩溃，或在错误决定情况下触发触发器。

- 修复 `ALTER COLUMN TYPE` 正确处理继承的检查约束(Pavan Deolasee)

  这在以前8.4版本中正常工作，并且现在在8.4及以后版本中也正常工作。

- 修复 `ALTER EXTENSION SET SCHEMA` 的错误以移动一些附属对象到新模式中(Álvaro Herrera, Dimitri Fontaine)
- 在扩展查询协议中正确处理 `CREATE TABLE AS EXECUTE` (Tom Lane)
- 不要在 `DROP RULE IF NOT EXISTS` and `DROP TRIGGER IF NOT EXISTS` 中修改输入解析树 (Tom Lane)

  如果重新执行这些类型之一的已缓存语句，那么可能产生错误。

- 修复 `REASSIGN OWNED` 处理表空间上的授权(Álvaro Herrera)
- 忽略视图系统列错误的 `pg_attribute` 项(Tom Lane)

  视图没有任何系统列。然而，当转换表到视图时我们忘了 删除该项。在9.3以及以后被正确修复，但在以前的分支中我们需要防卫 现有的错误转换视图。

- 修复规则输出以正确转储 `INSERT INTO` `_table_` DEFAULT VALUES(Tom Lane)
- 当在一个查询中有太多 `UNION` / `INTERSECT` / `EXCEPT` 子句时，防止堆栈溢出(Tom Lane)
- 当使用-1区分尽可能低的整数值时，避免平台相关错误(Xi Wang, Tom Lane)
- 修复日期分析中可能访问以前的字符串末尾(Hitoshi Harada)
- 如果在检查点期间发生XID概括并且 `wal_level` 是 `hot_standby`， 那么修复错误以提前XID时代(Tom Lane, Andres Freund)

  当这个错误对PostgreSQL自身没有特别的影响时， 对于依赖于 `txid_current()` 和相关函数是一个坏的应用：TXID值将出现回退。

- 修复 `pg_terminate_backend()` 和 `pg_cancel_backend()` 不要为非存在的目标过程抛出错误 (Josh Kupershmidt)

  当通过超级用户调用的时候，该种情况已经按照预期进行，但是当通过普通用户调用的时候并不这样。

- 修复页面边界 `pg_stat_replication`.`sync_state` 的显示(Kyotaro Horiguchi)
- 如果为了Unix域套接字的路径名长度超过特定平台限制， 那么产生可理解的错误消息 (Tom Lane, Andrew Dunstan)

  以前这可能导致一些无用的东西，比如"域名解析不可恢复故障"

- 当发送复合列值给客户端时，修复内存泄露(Tom Lane)
- 通过提交时不搜索子事务锁节省一些周期(Simon Riggs)

  在事务中持有许多排他锁，这些无效活动可能是相当昂贵的。

- 使得pg_ctl关于读取 `postmaster.pid` 文件更加健壮(Heikki Linnakangas)。

  这将修复竞争条件和可能的文件描述符泄漏。

- 如果提出错误编码数据，并且 `client_encoding` 设置是客户端编码，比如SJIS， 那么可能在psql中崩溃(Jiang Guiqing)

- 在数据不是归档预先数据段中使得pg_dump备份 `SEQUENCE SET` 项(Tom Lane)

  这种修复了 `--data-only` 和 `--section=data` 意义之间不良的不一致， 并修复备份被标记为可扩展配置表序列。

- 修复 `--clean` 模式中 `DROP DATABASE` 命令的pg_dump的处理(Guillaume Lelarge)

  9.2.0开始， `pg_dump --clean` 提出 `DROP DATABASE` ， 根据使用场景这是无用的或者危险的。现在不再是这样了。 这种变化也将修复 `--clean` 和 `--create` 的结合以正常运行，即，发出 `DROP DATABASE` 然后重新连接 目标数据库之前发出 `CREATE DATABASE` 。

- 为了循环依赖的视图和没有关系的选项修复pg_dump(Tom Lane)

  当视图涉及没有选项不正常工作的情况下的循环依赖时， 先前修复是备份关系选项。它发出 `ALTER VIEW foo SET ()` ， 这是无效的语法。

- 修复通过 `tar` 输出格式中pg_dump发出的 `restore.sql` 脚本中的错误(Tom Lane)

  该脚本在名字包括大写字母的表上可能失败。同时， 使脚本在 `--inserts` 模式中和规则COPY模式中能够恢复数据。

- 修复pg_restore接受符合POSIX标准的 `tar` 的文件(Brian Weaver, Tom Lane)

  pg_dump的 `tar` 输出模式的原始编码产生不能与POSIX标准完全一致的文件。 这是9.3版本的修正。这个补丁更新以前的分支，以致于它们会接受 不正确的和正确的格式，为了避免9.3出现的 兼容性问题。

- 修复通过pg_basebackup发出的 `tar` 文件到符合POSIX标准(Brian Weaver, Tom Lane)

- 当给出了数据目录相应路径时，修复pg_resetxlog以正确定位 `postmaster.pid` (Tom Lane)

  这个错误可能导致pg_resetxlog没有注意到使用数据目录的一个活跃postmaster。

- 修复libpq的 `lo_import()` 和 `lo_export()` 函数以正确报告文件I/O错误(Tom Lane)

- 修复嵌套结构指针变量的ecpg处理(Muhammad Usama)

- 修复ecpg的 `ecpg_get_data` 函数以正确处理数组(Michael Meskes)

- 防止pg_upgrade试图处理系统目录的TOAST表(Bruce Momjian)

当 `information_schema` 已被删除或重新创建时，这修复了发现的错误。 其他错误也是可能的。

- 通过设置新群集中 `synchronous_commit` 到 `off` 提升pg_upgrade性能(Bruce Momjian)
- 使得 `contrib/pageinspect` 的btree页检查函数当检查页时采用缓冲锁(Tom Lane)
- 解决 `malloc(0)` 和 `realloc(NULL, 0)` 不可移植操作(Tom Lane)

  平台上这些调用返回 `NULL`，一些代码错误地认为内存不足。 对数据库不包含用户自定义聚合已损坏的pg_dump是已知的。 可能还有其他的情况。

- 确保 `make install` 为扩展创建 `extension` 安装目录(Cédric Villemain)

  以前，如果在扩展的Makefile中设置 `MODULEDIR`，可以省略该步。

- 修复pgxs支持AIX上编译可加载模块(Tom Lane)

  编译不在AIX上运行的初始源码树外部模块。

- 为了Cuba, Israel, Jordan, Libya, Palestine, Western Samoa以及Brazil区域中DST变化规律更 新时区数据文件到tzdata版本2012j。

E.7. 版本9.2.1

> 发布日期: 2012-09-24

该版本包含来自9.2.0的各种修复。关于9.2主要版本新功能的信息，请参阅Section E.8。

E.7.1. 迁移到版本9.2.1

为了运行9.2.X不需要转储/恢复。

然而，你可能需要执行 `REINDEX` 和/或者 `VACUUM` 操作从下面第一个changelog项描述的数据损坏错误影响中恢复。

E.7.2. 变化

- 修复WAL回放期间共享缓冲区持久性标记(Jeff Davis)

  这个错误会导致在检查点缓冲区不被输出， 如果服务器稍后崩溃而没有输出这些缓冲，导致数据损坏。 损坏可能在随后崩溃恢复的任何服务器上发生， 但它很显然可能在备用从属服务器上发生， 因为这些执行了更多WAL回放。 btree和GIN索引有低概率损坏。 表"可见视图"有更高概率的损坏， 这可能会导致来自索引扫描的错误结果，正确的表数据不能被这错误损坏。

  虽然没有索引的损坏，由于这个错误已经在该字段发生， 作为预防措施建议产品升级到9.2.1后在合适的时间安装 `REINDEX` 所有btree和GIN索引。

  同时，当 `vacuum_freeze_table_age` 设置为零的时候， 建议执行所有表的 `VACUUM` ， 这将修复任何不正确可见性的视图数据。 当它需要更长的时间来完成的时候，可以调整 `vacuum_cost_delay` 以减少清理的性能影响。

- 修复涉及 `WHERE` `_indexed_column_` IN (`_list_of_values_`)查询输出可能不正确的排序(Tom Lane)

- 修复涉及 `GROUP BY` 表达式和window函数和聚集函数的查询的规划器错误(Tom Lane)

- 修复执行器参数的规划器分配(Tom Lane)

  该错误可能导致来自扫描同一个 `WITH` 子查询多次的查询错误结果。

- 提高索引扫描中join条件的规划器处理(Tom Lane)

- 提高涉及前缀的文本搜索查询的选择性估计，比如 `_word_``:*` 模式(Tom Lane)

- 修复权限变化的延迟识别(Tom Lane)

  除了有可能没有注意到自事务开始时已提交的并行 `GRANT` 或者 `REVOKE` 的事务外，不需要锁的命令。

- 当列是数组类型域时，修复 `ANALYZE` 而不失败(Tom Lane)

- 如果递归PL/Perl函数在执行的时候被重新定义，那么防止PL/Perl崩溃(Tom Lane)

- 解决PL/Perl中可能的错误优化(Tom Lane)

  一些Linux发布包含 `pthread.h` 不正确的版本，导致PL/Perl中不正确的编译代码， 如果PL/Perl函数调用另外抛出错误的一个，那么导致崩溃。

- 删除来自pg_upgrade的pg_config上的不必要依赖(Peter Eisentraut)

- 为了Fiji的DST变化规律更新时区数据文件到tzdata版本2012f

E.8. 版本9.2

> 发布日期: 2012-09-10

E.8.1. 概述

此版本主要集中在性能上的改进，但新的SQL功能不缺乏。工作在备份支持领域继续进行。主要功能包括：

- 允许查询从索引中检索数据，避免堆访问(索引扫描)
- 即使当使用预备语句时，允许规划器为特定参数值产生自定义规划
- 提高规划器使用嵌套循环内部索引扫描的能力
- 允许流复制制转发数据到其他从属(级联复制)
- 允许pg_basebackup执行来自备用服务器的基础备份
- 添加pg_receivexlog工具以归档作为他们写入的WAL文件
- 添加SP-GiST (空间划分GiST)索引访问方法
- 添加range data types支持
- 添加 `JSON` 数据类型
- 为视图增加 `security_barrier` 选项
- 允许libpq连接字符串有URI的格式
- 添加单行处理模式到libpq 更好的处理大的结果集

以上项的更多详情在下面的章节有介绍。

E.8.2. 迁移到版本9.2

使用pg_dump转储/恢复，或者使用pg_upgrade 从任何以前版本中迁移数据。

版本9.2包含可能影响与先前版本的兼容性的一些变化。 观察下面的不兼容：

E.8.2.1. 系统目录

- 从 `pg_tablespace` 中删除 `spclocation` 字段(Magnus Hagander)

该字段实际上定义表空间位置的符号链接的复制，从而当移动表空间时，可能有遗漏错误风险。 在服务器关闭时，通过手动调节符号链接，这种变化允许删除表空间目录。 为了替换该字段，我们添加 pg_tablespace_location() 允许符号链接查询。

- 移动 tsvector 最常见元素统计到新的 pg_stats 列(Alexander Korotkov)

  为了在 most_common_vals 和 most_common_freqs 的 tsvector 列中可用的原先的数据 查询 most_common_elems 和 most_common_elem_freqs 。

E.8.2.2. 函数

- 删除hstore's => 操作符(Robert Haas)

  用户应该使用 hstore(text, text) 。 自PostgreSQL 9.0， 当创建命名 => 的操作符，则发出一个警告信息。 因为SQL标准保留了另一个使用的标记。

- 确保 xpath() 在字符串值中逃逸特殊字符(Florian Pflug)

  倘若没有这个，那么对于结果可能不是有效的XML。

- 使用 pg_relation_size() 并且如果该对象不存在则返回空(Phil Sorber)

  这可以防止查询调用这些函数从并发 DROP 后立即返回错误。

- 使用 EXTRACT(EPOCH FROM *timestamp without time zone*) 从当地午夜测量时间，而不是UTC午夜(Tom Lane)

  这一变化恢复在版本7.3中缺乏考虑的变化。 从UTC午夜测量是不一致的， 因为它使结果依赖于 timezone 设置， 这不应该有 timestamp without time zone 的计算。以前的操作仍然可以通过映射输入 值到 timestamp with time zone 可用。

- 正确解析带有尾随 yesterday , today 和 tomorrow 的时间字符串(Dean Rasheed)

  先前 SELECT '04:00:00 yesterday'::timestamp 返回午夜时yesterday的日期。

- 修复 to_date() 和 to_timestamp() 以封装不完整日期到2020(Bruce Momjian)

  先前，提供年以及不一致封装低于四位的年掩码。

E.8.2.3. 对象修改

- 防止 ALTER DOMAIN 工作于非域类型上(Peter Eisentraut)

  在非域类型上所有者和模式变化是可能的。

- 在 CREATE FUNCTION 中 不再强制小写过程语言名字(Robert Haas)

当反引号语言标识符仍然是小写字母时，字符串和带引号的标识符不再强行向下。 因此，例如 `CREATE FUNCTION ... LANGUAGE 'C'` 将不再工作； 它必须拼写 `'c'`，或更好地省略引号。

- 改变外键执行触发器系统产生的名称(Tom Lane)

  这一变化确保在涉及自我参照的外键约束的情况下以正确的顺序触发触发器。

E.8.2.4. 命令行工具

- 提供一致的反引号，变量扩张， 以及psql元命令参数中的引用子字符串操作(Tom Lane)

  以前，当不通过相邻文本空白分隔的时候。奇怪地处理这种引用。 例如 `'FOO'BAR` 作为 `FOO BAR` 被输出 （空间意外插入）并且 `FOO'BAR'BAZ` 输出不变 （不删除期望的引用）。

- 不再将clusterdb表名看作双引号； 不再将reindexdb表和索引名看作双引号(Bruce Momjian)

  如果希望引用，那么用户现在必须在命令参数中包含双引号。

- createuser缺省不再提示选项设置(Peter Eisentraut)

  使用 `--interactive` 获得旧的操作。

- 禁用dropuser中的用户名提示， 除非指定 `--interactive` (Peter Eisentraut)

E.8.2.5. 服务器设置

- 添加服务器参数以指定服务器端SSL文件位置

  这允许改变名字和文件位置，该文件先前作为 `server.crt`， `server.key`， `root.crt` 和数据目录中 `root.crl` 的硬编码。缺省时服务器将不再检查 *`root.crt`* 或者 *`root.crl`* ；加载这些文件，相关参数必须设置为非缺省值。

- 删除 `silent_mode` 参数(Heikki Linnakangas)

  可以使用 `pg_ctl start -l postmaster.log` 获得类似操作。

- 删除不再需要的 `wal_sender_delay` 参数(Tom Lane)

- 删除 `custom_variable_classes` 参数(Tom Lane)

  通过该设置提供的检查是含糊的。现在可以以任何类名作为任何设置前缀。

E.8.2.6. 监控

- 重命名 `pg_stat_activity` `.procpid` 到 `pid`， 匹配其他系统表(Magnus Hagander)
- 创建一个单一的 `pg_stat_activity` 列以报告进程状态(Scott Mead, Magnus Hagander)

  先前的 `query` 和 `query_start` 值现在仍然可用于空闲会话， 允许增强分析。

- 当查询完成时，重命名 `pg_stat_activity` . `current_query` 到 `query` ， 因为它是不清楚的(Magnus Hagander)
- 改变所有SQL级别统计时序值到以毫秒计的 `float8` 列

  这种变化消除设计假设该值精确到微秒，没有更多的（因为 `float8` 值可以是分数）。 受影响的列： `pg_stat_user_functions` . `total_time` , `pg_stat_user_functions` . `self_time` , `pg_stat_x` . `total_time` , 和 `pg_stat_xact_user_functions` . `self_time` 。 潜在这些列的统计函数现在返回 `float8` 毫秒， 而不是 `bigint` 微秒。现在以毫秒测量 `contrib/pg_stat_statements` 的 `total_time` 列。

E.8.3. 变化

下面你将发现在PostgreSQL 9.2和先前主要版本之间变化的详细情况。

E.8.3.1. 服务器

E.8.3.1.1. 性能

- 允许查询只从索引中检索数据，避免堆访问(Robert Haas, Ibrar Ahmed, Heikki Linnakangas, Tom Lane)

  此功能通常被称为只索引扫描。 为了只包含对所有会话可见的元组堆页可以忽略堆访问， 正如可见视图报道的；因此该效益主要适用于大多数静态数据。可见视图将损坏安全作为 执行该功能的必要组成部分。

- 增加SP-GiST(空间划分GiST)索引访问方法(Teodor Sigaev, Oleg Bartunov, Tom Lane)

  SP-GiST与GIST的灵活性相比较，但支持不平衡分段搜索结构而不是平衡树。 为了适当的问题，SP-GiST在索引编译时间和搜索时间上比GiST更快。

- 允许组提交在重负载下有效工作(Peter Geoghegan, Simon Riggs, Heikki Linnakangas)

  此前，批量提交无效作为已增加的写入工作量，由于内部锁争用。

- 允许使用一个新的快速路径锁定机制管理未竞争锁(Robert Haas)
- 减少创建虚拟事务ID锁的开销(Robert Haas)
- 减少串行化隔离级别锁的开销(Dan Ports)

- 提高PowerPC和Itanium spinlock性能(Manabu Ori, Robert Haas, Tom Lane)
- 减少共享的无效缓存消息开销(Robert Haas)
- 移动 `PGPROC` 共享内存数组频繁访问的成员到一个单独数组中(Pavan Deolasee, Heikki Linnakangas, Robert Haas)
- 通过成批地添加元组到堆中提高 `COPY` 性能(Heikki Linnakangas)
- 通过产生内存分配开销少的树的几何数据类型提高GiST索引性能。
- 提高GiST索引编译时间(Alexander Korotkov, Heikki Linnakangas)
- 允许提示位被迅速设置为临时的和未标记表(Robert Haas)
- 允许通过内联进行排序，非SQL可调用比较函数(Peter Geoghegan, Robert Haas, Tom Lane)
- 增大基于 `shared_buffers` 的CLOG缓冲区规模数量(Robert Haas, Simon Riggs, Tom Lane)
- 当删除表或者数据库时，提高发生的缓冲池扫描性能(Jeff Janes, Simon Riggs)
- 当许多表被删除或截断时，提高检查点的fsync请求阵列性能(Tom Lane)
- Windows上传递文件描述符安全码到子进程(Heikki Linnakangas)

  这允许Windows会话使用比以前更多的打开文件描述符。

E.8.3.1.2. 进程管理

- 创建一个执行检查点的专门后台进程(Simon Riggs)

  原先后端写进程执行脏页写入和检查点。分离成两个过程允许每个目标可预见地完成。

- 通过快速唤醒walwriter提高异步提交操作(Simon Riggs)

  以前， `wal_writer_delay` 触发WAL冲洗磁盘； 现在填充WAL缓冲区也触发WAL写入。

- 在不活跃期间让bgwriter, walwriter， 检查点，数据采集器，日志收集器和归档日志后端进程更有效睡眠(Peter Geoghegan, Tom Lane)

  当没什么事可做的时候，这一系列的变化会降低进程唤醒频率，大幅降低空闲服务器上的功耗。

E.8.3.1.3. 优化器

- 允许规划器生成特定参数的自定义规划，即使当使用预处理语句的时候(Tom Lane)

以往，一个预备语句总是有一个单一的"通用"计划被用于所有的参数值， 通常远不如用于包含显式恒定值的非预备语句规划。现在，规划器 试图生成特定参数值的自定义规划。 一个通用计划将只能用在自定义规划多次证明没有任何好处的情况。 这种变化应该消除原先从预备语句使用中见到过的性能损失（包括PL/pgSQL中的非动态语句）。

- 提高规划器使用嵌套循环内部索引扫描的能力(Tom Lane)

  新的"参数化路径"机制允许内部索引扫描使用超过扫描中一个连接水平的关系值。 这可以大大提高语义限制（如外连接）允许的连接顺序的情况下的性能。

- 提高对外部数据封装器的规划API (Etsuro Fujita, Shigeru Hanada, Tom Lane)

  封装器现在可以为表提供多种访问"路径"，在连接规划中更加灵活。

- 识别非表关系的自相矛盾限制分句(Tom Lane)

  当 `constraint_exclusion` 打开 时，执行 这个检查。

- 允许 `indexed_col op ANY(ARRAY[...])` 条件用于纯索引扫描并且仅仅索引扫描(Tom Lane)

  以前这样的条件只能用于位图索引扫描。

- 在 `boolean` 列上支持 `MIN` / `MAX` 索引优化(Marti Raudsepp)
- 当估计设置行数时，解释了在 `SELECT` 目标列中设置返回函数的原因(Tom Lane)
- 修复规划器更可靠的处理带有重复列索引(Tom Lane)
- 收集并且使用数组元素次数统计(Alexander Korotkov, Tom Lane)

  这种变化提高了数组 `<@` ， `&&` 和 `@>` 操作符（数组容量和重叠部分）的选择性估计。

- 允许收集外表的统计(Etsuro Fujita)
- 提高使用部分索引的成本估计(Tom Lane)
- 提高在子查询中引用列统计的规划器能力(Tom Lane)
- 提高子查询使用 `DISTINCT` 的统计估计(Tom Lane)

E.8.3.1.4. 认证

- 不要把角色名字和 `pg_hba.conf` 中 声明的 `samerole` 看作自动包含超级用户的(Andrew Dunstan)

  这使得它更容易使用组角色的 `reject` 行。

- 调节 `pg_hba.conf` 过程更加一致地处理令牌解析(Brendan Jurd, Álvaro Herrera)
- 不允许空 `pg_hba.conf` 文件(Tom Lane)

这样做是为了更快速检测配置错误。

- 使用超级用户权限意味着复制权限(Noah Misch)

  这样避免了需要明确地分配这类权限。

E.8.3.1.5. 监控

- 试图在后端崩溃期间记录当前查询字符串(Marti Raudsepp)
- 使得自动清理I/O活动日志更加冗长(Greg Smith, Noah Misch)

  这条记录是通过 `log_autovacuum_min_duration` 触发的。

- 使WAL回放尽快报告错误(Fujii Masao)

  一旦服务器到主模式只报告错误的情况。

- 添加 `pg_xlog_location_diff()` 以简化WAL位置比较(Euler Taveira de Oliveira)

  这对于计算复制滞后是有用的。

- 支持Windows上可配置的事件日志应用程序名称(MauMau, Magnus Hagander)

  这允许不同的情况下使用不同的标识符事件日志， 通过设置 `event_source` 服务器参数，这类似于 `syslog_ident` 工作方式。

- 改变"意外的块结束"消息到 `DEBUG1` 级别，除非有一个已打开事务(Magnus Hagander)

  这种变化减少笨拙地关闭数据库连接应用导致的日志振动。

E.8.3.1.6. 统计视图

- 跟踪临时文件大小和 `pg_stat_database` 系统视图中的文件数(Tomas Vondra)
- 添加一个死锁计数器到 `pg_stat_database` 系统视图(Magnus Hagander)
- 添加服务器参数 `track_io_timing` 跟踪I/O时序(Ants Aasma, Robert Haas)
- 报告 `pg_stat_bgwriter` 中的检查点定时信息(Greg Smith, Peter Geoghegan)

E.8.3.1.7. 服务器设置

- 默默忽略 `search_path` 指定的不存在的模式(Tom Lane)

  这使得更加方便使用通用路径设置，可能包含一些不存在于数据库中的模式。

- 允许超级用户设置 `deadlock_timeout` 每个会话， 而不是每个集群(Noah Misch)

  这允许为了可能涉及到死锁的事务减少 `deadlock_timeout` ， 因此更快地检查故障。 另外，增加该值可以用于减少由于死锁选择取消会话机会。

- 添加服务器参数 `temp_file_limit` 限制每个会话临时文件空间使用率(Mark Kirkwood)
- 在加载相关扩展之前允许超级用户 `SET` 扩展的超级用户自定义变量(Tom Lane)

  系统现在记得是否 `SET` 通过超级用户被执行， 因此当加载扩展时，可以执行适当的权限检查。
- 添加postmaster `-C` 选项查询配置参数(Bruce Momjian)

  允许pg_ctl更好地处理 `PGDATA` 或者 `-D` 指向配置目录的情况。
- 以 `CREATE DATABASE` 中隐含值替换空的区域名(Tom Lane)

  这可以防止服务器重启后不同地解释 `pg_database` . `datcollate` 或者 `datctype` 的情况。

E.8.3.1.7.1. `postgresql.conf`

- 允许报告 `postgresql.conf` 中的多个错误， 而不仅仅是第一个(Alexey Klyukin, Tom Lane)
- 允许通过所有会话处理 `postgresql.conf` 的加载，即使有一些设置对特定会话是无效的(Alexey Klyukin)

  以前，这种无效会话值可能导致所有设置变化被该会话忽略。
- 为配置文件添加一个 `include_if_exists` 功能(Greg Smith)

  这个和 `include` 一样运行，除了如果忽略该文件不会抛出错误外。
- 在initdb期间识别服务器时区，并且相应设置 `postgresql.conf` 项 `timezone` 和 `log_timezone`

  这避免了服务器启动时昂贵的时区探测。
- 修复 `pg_settings` 以 报告Windows上 `postgresql.conf` 行号(Tom Lane)

E.8.3.2. 备份和恢复

- 允许流复制转发数据到其他从属(级联复制)(Fujii Masao)

  以前只有主服务器可以提供流复制日志文件到备用服务器。
- 添加新的 `synchronous_commit` 模式 `remote_write` (Fujii Masao, Simon Riggs)

  该模式等待备用服务器写事务数据到它自己的操作系统，但是不等待数据被刷新到备用磁盘。
- 添加pg_receivexlog工具 归档他们写入的WAL文件变化，而不是等待完成的WAL文件(Magnus Hagander)

- 允许pg_basebackup从 备用服务器上做基础备份(Jun Ishizuka, Fujii Masao)

  该功能允许基础备份工作从主服务器上被卸载。

- 当pg_basebackup在执行备份时，允许WAL文件流(Magnus Hagander)

  允许在主库上丢弃之前传递WAL文件到备库。

E.8.3.3. 查询

- 如果断开客户端连接，取消运行的查询(Florian Pflug)

  如果在查询中后端监测客户端连接损坏，那么它现在将取消查询而不是尝试完成它。

- 在行表达式运行时保留列名(Andrew Dunstan, Tom Lane)

  当行值被转换为 `hstore` 或者 `json` 类型时：结果值字段有期望名，那么该变化允许有更好结果。

- 改善用于子- `SELECT` 结果的列标签(Marti Raudsepp)

  以前使用一般标签 `?column?` 。

- 提高决定未知值类型的探索法(Tom Lane)

  当考虑多态操作符的时候， 长期规则是一个未知常量可能与现在应用它的操作者一端的值有同样类型， 而不仅仅操作符匹配。

- 关于创建映射或者来自域类型的警告(Robert Haas)

  这种投射没有影响。

- 当行不能进行 `CHECK` 或者 `NOT NULL` 约束时， 显示行内容作为错误详细信息 (Jan Kundrát)

  当插入或者更新处理多行时， 这将很容易识别哪行是有问题的。

E.8.3.4. 对象操作

- 在并行DDL中提供更可靠操作(Robert Haas, Noah Misch)

  该变化增加了锁定，它应该删除多种情况中"高速缓存查找失败"错误。 另外，不再可能添加关系到正在删除的模式中，先前导致不一致系统目录内容的情况。

- 添加 `CONCURRENTLY` 选项到 `DROP INDEX` (Simon Riggs)

  允许索引删除不锁定其他会话。

- 允许外数据包有每一列选项(Shigeru Hanada)

- 改进视图定义的输出(Andrew Dunstan)

E.8.3.4.1. 约束

- 允许 `CHECK` 约束被声明为 `NOT VALID` (Álvaro Herrera)

  添加 `NOT VALID` 约束不会导致扫描的表可以验证已存在行满足该约束。 随后，检查新添加或者更新的行。当考虑 `constraint_exclusion` 时， 规划器忽略这些约束。因为它不确定所有行可以满足该约束。

  新的 `ALTER TABLE VALIDATE` 命令允许为已存在 行检查 `NOT VALID` 限制，之后被转换成普通约束。

- 允许 `CHECK` 限制被声明为 `NO INHERIT` (Nikhil Sontakke, Alex Hunsaker, Álvaro Herrera)

  这使得它们只在父表是可执行的，而不是子表。

- 添加该能力到rename约束(Peter Eisentraut)

E.8.3.4.2. `ALTER`

- 减少重新编译表需要以及 索引某个 `ALTER TABLE` ... `ALTER COLUMN TYPE` 操作(Noah Misch)

  增加 `varchar` 或者 `varbit` 列长度限制，或者完全删除限制，不再需要表重写。 类似地，增加 `numeric` 列允许的精度，或者改变约束的 `numeric` 列到不受约束的 `numeric` ， 不再需要表重写。 在涉及 `interval` ， `timestamp` 和 `timestamptz` 类型的相似情况中避免表重写。

- 避免 `ALTER TABLE` 在 不必要的情况下验证外键约束(Noah Misch)

- 增加 `IF EXISTS` 选项到一些 `ALTER` 命令中(Pavel Stehule)

  比如 `ALTER FOREIGN TABLE IF EXISTS foo RENAME TO bar` 。

- 添加 `ALTER FOREIGN DATA WRAPPER` ... `RENAME` 和 `ALTER SERVER` ... `RENAME` (Peter Eisentraut)

- 添加 `ALTER DOMAIN` ... `RENAME` (Peter Eisentraut)

  你可能已经使用 `ALTER TYPE` 重命名域。

- 在不存在约束上为 `ALTER DOMAIN` ... `DROP CONSTRAINT` 抛出一个错误(Peter Eisentraut)

  一个 `IF EXISTS` 选项已被添加以提供先前操作。

E.8.3.4.3. `CREATE TABLE`

- 允许 `CREATE TABLE (LIKE ...)` 来自外表，视图和复合类型(Peter Eisentraut)

比如，模式匹配视图的允许创建表。

- 当拷贝索引注释时，修复 `CREATE TABLE (LIKE ...)` 以避免索引名冲突(Tom Lane)
- 修复 `CREATE TABLE` ... `AS EXECUTE` 以处理 `WITH NO DATA` 和列名称规格(Tom Lane)

E.8.3.4.4. 对象权限

- 为视图添加 `security_barrier` 选项(KaiGai Kohei, Robert Haas)

  该选项可以防止优化可能允许视图受保护数据暴露给用户， 比如推动涉及不安全函数的子句到视图的 `WHERE` 子句。 这种视图预期比普通视图执行更差。

- 添加新的 `LEAKPROOF` 函数属性以标记函 数可以安全地向下推进到 `security_barrier` 视图(KaiGai Kohei)
- 增加数据类型权限支持(Peter Eisentraut)

  在类型和域上添加SQL-一致 `USAGE` 权限支持。 其意图是可以限制哪个用户在类型上可以创建依赖，因为这种依赖 限制修改该类型的用户能力。

- 检查在 `SELECT INTO` / `CREATE TABLE AS` 中的 `INSERT` 权限(KaiGai Kohei)

  因为通过 `SELECT INTO` 或者 `CREATE TABLE AS` 创建该对象，那么创建者通常有插入权限；但是有一种不真的困境情况， 比如当 `ALTER DEFAULT PRIVILEGES` 已经删除该权限时。

E.8.3.5. 实用操作

- 允许 `VACUUM` 更容易忽略不能被锁定的页(Simon Riggs, Robert Haas)

  这种变化会大大降低 `VACUUM` 获得"stuck"等待其他会话的发生率。

- 使得 `EXPLAIN` `(BUFFERS)` 计算脏块和写入的(Robert Haas)
- 使得 `EXPLAIN ANALYZE` 报告通过过滤步骤拒绝的行数(Marko Tiikkaja)
- 当时间值是不想要的，允许 `EXPLAIN ANALYZE` 避免时间开销(Tomas Vondra)

  通过设置新的 `TIMING` 选项到 `FALSE` 来完成。

E.8.3.6. 数据类型

- 添加支持range data types (Jeff Davis, Tom Lane, Alexander Korotkov)

  一系列数据类型存储从属于它的基本数据类型的上限和下限。 它支持类似包含，重叠和交叉的操作。

- 添加 `JSON` 数据类型(Robert Haas)

这种类型存储带有适当验证的JSON (JavaScript对象表示法)数据。

- 添加 `array_to_json()` 和 `row_to_json()` (Andrew Dunstan)
- 添加 `SMALLSERIAL` 数据类型(Mike Pultz)

  这就像 `SERIAL`，除了它以两个字节整数列(`int2`)存储序列之外。

- 允许domains被 声明为 `NOT VALID` (Álvaro Herrera)

  在域创建时间设置该选项，或者通过 `ALTER DOMAIN` ... `ADD CONSTRAINT` ... `NOT VALID` . `ALTER DOMAIN` ... `VALIDATE CONSTRAINT` 充分验证该限制。

- 为了 `money` 数据类型支持更多的区域指定格式选项(Tom Lane)

  特别的，为了该值顺序，标志，以及货币输出中的货币符号纪念POSIX选项。另外，确保千位分隔符只被插入到小数点左边，正如POSIX要求的。

- 为 `macaddr` 数据类型添加按位"and", "or"和"not"操作符(Brendan Jurd)
- 当提供一个标量值时， 允许 `xpath()` 返回 单一元素XML数组(Florian Pflug)

  先前，它返回空数组。这种变化也将导致 `xpath_exists()` 为这种表达式返回真，而不是假。

- 提高XML错误处理以变得更健壮(Florian Pflug)

E.8.3.7. 函数

- 允许非超级用户在其他会话从属于同一用户时使用 `pg_cancel_backend()` 和 `pg_terminate_backend()` (Magnus Hagander, Josh Kupershmidt, Dan Farina)

  以前只有超级用户被允许使用这些函数。

- 允许事务快照的输入和输出(Joachim Wieland, Tom Lane)

  这允许多个事务共享数据库状态的同一视图。快照是通过 `pg_export_snapshot()` 输出，通过 `SET TRANSACTION SNAPSHOT` 输入。只有当前正在运行的事务快照可以被输入。

- 支持表达式上的 `COLLATION FOR` (Peter Eisentraut)

  返回表达式排序规则的字符串表示。

- 添加 `pg_opfamily_is_visible()` (Josh Kupershmidt)
- 添加 `numeric` variant of `pg_size_pretty()` 适用于 `pg_xlog_location_diff()` (Fujii Masao)
- 添加 `pg_trigger_depth()` 函数(Kevin Grittner)

报告当前触发器调用深度。

- 允许 `string_agg()` 处理 `bytea` 值(Pavel Stehule)
- 在一个较大的量化子表达式中发生后向引用的地方修复正则表达式(Tom Lane)

  比如 `^(\w+)( \1)+$` 。先前版本并不检查后向引用实际匹配第一次出现。

E.8.3.8. 信息模式

- 添加信息模式视图 `role_udt_grants` , `udt_privileges` , 和 `user_defined_types` (Peter Eisentraut)
- 添加复合类型属性到信息模式 `element_types` 视图(Peter Eisentraut)
- 信息模式中实现 `interval_type` 列(Peter Eisentraut)

  以前这些列读取为空。

- 在信息模式 `attributes` , `columns` , `domains` 和 `element_types` 视图中实现排序规则相关列(Peter Eisentraut)
- 在信息模式 `table_privileges` 视图中实现 `with_hierarchy` 列 (Peter Eisentraut)
- 增加序列 `USAGE` 权限显示到信息模式中(Peter Eisentraut)
- 使信息模式显示缺省权限(Peter Eisentraut)

  先前，非空缺省权限没有出现在视图中。

E.8.3.9. 服务器端语言

E.8.3.9.1. PL/pgSQL服务器端语言

- 允许PL/pgSQL `OPEN` 游标命令提供参数名(Yeb Havinga)
- 添加 `GET STACKED DIAGNOSTICS` PL/pgSQL命令检索异常信息(Pavel Stehule)
- 通过缓存类型信息加快PL/pgSQL数组赋值(Pavel Stehule)
- 提高性能以及为长连续 `ELSIF` 子句内存损耗(Tom Lane)
- 在PL/pgSQL错误消息中输出函数签名，而不仅仅是名字(Pavel Stehule)

E.8.3.9.2. PL/Python服务器端语言

- 添加PL/Python SPI游标支持 (Jan Urbanski)

  这允许PL/Python读取部分结果集。

- 添加结果元数据函数到PL/Python (Peter Eisentraut)

  具体地说，这增加了结果对象函数 `.colnames`，`.coltypes` 和 `.coltypmods`。

- 删除支持Python 2.2 (Peter Eisentraut)

E.8.3.9.3. SQL服务器端语言

- 允许SQL语言函数参照参数名(Matthew Draper)

  为了使用这个，仅仅命名函数参数，并且然后参考SQL 函数体中的参数名。

E.8.3.10. 客户端应用

- 添加initdb 选项 `--auth-local` 和 `--auth-host` (Peter Eisentraut)

  这允许 `local` 和 `host` `pg_hba.conf` 认证设置的分散控制， `--auth` 仍然控制着两个。

- 添加 `--replication` / `--no-replication` 标记到 createuser以控制备份权限(Fujii Masao)
- 添加 `--if-exists` 选项到 dropdb和 dropuser (Josh Kupershmidt)
- 给出命令行工具以指定要连接数据库名字， 如果 `postgres` 数据库连接失败， 那么回退到 `template1` (Robert Haas)

E.8.3.10.1. psql

- 添加基于显示宽度的显示模式以自动扩展输出(Peter Eisentraut)

  增加 `auto` 选项到 `\x` 命令，当正常 输出比屏幕宽的时候，会切换到扩展模式。

- 允许脚本文件的包含是相对于它被调用的文件目录命名的(Gurjeet Singh)

  这是执行新的命令 `\ir` 。

- 在psql变量名中添加支持非ASCII字符(Tom Lane)
- 添加支持主要版本特定 `.psqlrc` 文件(Bruce Momjian)

  psql支持次要版本特定 `.psqlrc` 文件。

- 提供环境变量覆盖psql历史并且启动文件位置(Andrew Dunstan)

  如果设置，那么 `PSQL_HISTORY` 和 `PSQLRC` 决定这些文件名。

- 添加 `\setenv` 命令修改传递给子进程的环境变量(Andrew Dunstan)
- 命名 `.sql` 扩展的psql的临时编辑文件(Peter Eisentraut)

  这允许扩展敏感编辑者选择正确模式。

- 允许psql使用零字节字段并且记录分隔符(Peter Eisentraut)

  使用零字节(NUL)分隔符的各种shell工具，比如find。

- 使用 `\timing` 选项报告查询失败时间(Magnus Hagander)

  以前只为成功查询报道时间。

- 统一并固定 `\copy` 和SQL `COPY` 的psql处理(Noah Misch)

  这修复错误操作更加可预测并且设置 `\set ON_ERROR_ROLLBACK` 。

E.8.3.10.2. 信息命令

- 使得 `\d` 在序列上显示拥有它的表/列名(Magnus Hagander)
- 显示 `\d+` 中列的统计目标(Magnus Hagander)
- 显示 `\du` 中角色密码截止日期(Fabrízio de Royes Mello)
- 显示投射，转换，域和语言说明(Josh Kupershmidt)

  这些分别包含在 `\dC+`， `\dc+`, `\dD+` 和 `\dL` 输出中。

- 显示SQL/MED对象说明(Josh Kupershmidt)

  这些分别包括在 `\des+`, `\det+` 输出， `\dew+` 外服务器，外 表，和外数据包装器。

- 改变 `\dd` 仅显示没有自身反斜杠命令对象类型的说明(Josh Kupershmidt)

E.8.3.10.3. Tab实现

- 在psql tab实现中， 在大小写情况下依照新的 `COMP_KEYWORD_CASE` 设置 实现SQL关键字
- 添加tab实现支持 `EXECUTE` (Andreas Karlsson)
- 在 `GRANT` / `REVOKE` 中 添加角色引用的tab实现(Peter Eisentraut)
- 当必要的时候，允许文件名的tab实现可以提供引用(Noah Misch)
- 改变tab实现支持 `TABLE` 也包含视图(Magnus Hagander)

E.8.3.10.4. pg_dump

- 添加 `--exclude-table-data` 选项到 pg_dump (Andrew Dunstan)

  这允许在每个表基础上备份表的定义而不是它的数据。

- 添加 `--section` 选项到pg_dump 和pg_restore (Andrew Dunstan)

  有效值是 `pre-data` , `data` , 和 `post-data` 。给定的该选项不止一次的选择两个或更多部分。

- 使用 pg_dumpall首先备份 所有角色，然后角色上的所有配置设置(Phil Sorber)

  这允许角色的配置设置提及其他没有产生错误的角色。

- 如果在新的集群中丢失 `postgres` 数据库，那么允许 pg_dumpall避免错误(Robert Haas)

- 按照用户名顺序备份外服务器用户映射(Peter Eisentraut)

  这有助于产生确定性备份文件。

- 以可预见性顺序备份操作符(Peter Eisentraut)

- 当扩展配置表通过pg_dump被备份时，收紧规则(Tom Lane)

- 使得pg_dump发出更多有用的依赖信息(Tom Lane)

  包含在归档格式转储中的依赖关系以前使用非常有限， 因为他们经常引用似乎 不在转储中的对象。 现在他们在转储对象之间 代表实际的依赖关系（可能是间接的）。

- 当备份多个数据库对象时，提高pg_dump的性能(Tom Lane)

E.8.3.11. libpq

- 允许libpq连接 字符串有URI 的格式(Alexander Shulgin)

  该语法以 `postgres://` 开头。 这可以允许应用程序为URI表示数据库连接避免实现它们自身解析器。

- 添加连接选项 以禁用SSL压缩 (Laurenz Albe)

  这可以用于删除快速网络中SSL压缩的开销。

- 为了更好处理大的结果集添加 单行处理模式

  此前，libpq将 其返回给应用程序之前总是收集内存中整个查询结果。

- 添加 `const` 限定符 到函数 `PQconnectdbParams` , `PQconnectStartParams` , 和 `PQpingParams` 的声明(Lionel Elie Mamane)

- 允许 `.pgpass` 文件在密码域中包含转义字符(Robert Haas)

- 当必须终止进程时，尝试库函数使用 `abort()` 代替 `exit()` (Peter Eisentraut)

  这个选择并不妨碍使用正常退出代码程序， 并产生一个可以由调用者捕捉的信号。

E.8.3.12. 源码

- 删除封闭端口号(Peter Eisentraut)

  不再支持下列平台：dgux, nextstep, sunos4, svr4, ultrix4, univel, bsdi。

- 添加使用 MS Visual Studio 2010支持编译(Brar Piening)
- 启动使用MinGW-w64 32-位编译器进行编译(Lars Kanis)
- 在安装期间安装 `plpgsql.h` 到 `include/server` (Heikki Linnakangas)
- 提高闩锁装置以包含postmaster终止的检测(Peter Geoghegan, Heikki Linnakangas, Tom Lane)

  后端进程先前必须意识到调查事件，这消除了主要原因之一。
- 支持的地方使用C灵活数组元素(Peter Eisentraut)
- 提高并行事务回归测试(isolationtester) (Noah Misch)
- 在当前目录中修改thread_test创建它的测试文件，而不是 `/tmp` (Bruce Momjian)
- 提高flex和bison警告和错误报告(Tom Lane)
- 添加内存屏障支持(Robert Haas)

  这是目前未使用的。
- 修改pgindent使用typedef文件(Bruce Momjian)
- 由于被发送到服务器添加处理消息钩子(Martin Pihlak)
- 为 `DROP` 命令添加对象访问钩(KaiGai Kohei)
- 集中 `DROP` 处理一些对象类型(KaiGai Kohei)
- 添加pg_upgrade测试套件(Peter Eisentraut)
- 伴随TCL 8.5.11 同步正则表达式代码 并且改善内部处理(Tom Lane)
- 移动CRC表到libpgport， 在一个单独的include文件中提供它们(Daniel Farina)
- 为了用于主要发布声明 创建添加选项到git_changelog (Bruce Momjian)
- 支持Linux的 `/proc/self/oom_score_adj` API (Tom Lane)

E.8.3.13. 额外模块

- 通过使用libpq的新单行处理模式提高dblink的效率 (Kyotaro Horiguchi, Marko Kreen)

  这个改进不适用于 `dblink_send_query()` / `dblink_get_result()` 。
- 在file_fdw中支持 `force_not_null` 选项(Shigeru Hanada)
- 为pg_archivecleanup实现 演习模式(Gabriele Bartolini)

这只输出已删除文件名。

- 添加新的pgbench切换 `--unlogged-tables` , `--tablespace` 和 `--index-tablespace` (Robert Haas)

- 改变pg_test_fsync以测试 一定量的时间， 而不是固定周期数(Bruce Momjian)

  删除 `-o` /周期选项， 并且添加 `-s` /秒。

- 添加 pg_test_timing功能 以测量时钟一致性和时间开销 (Ants Aasma, Greg Smith)

- 添加tcn（触发变更通知） 模块在表变更上生成 `NOTIFY` 事件(Kevin Grittner)

E.8.3.13.1. pg_upgrade

- 调整pg_upgrade环境变量(Bruce Momjian)

  重命名data,bin和以 `PG` 开头的port环境变量， 支持 `PGPORTOLD` / `PGPORTNEW` , 取代 `PGPORT` 。

- 检查pg_upgrade记录和错误报告(Bruce Momjian)

  创建四个附加的日志文件，并且成功时删除它们。 添加 `-r` / `--retain` 选项无条件的保留这些文件。同时 删除不必要的pg_upgrade选项 `-g` / `-G` / `-l` 选项， 固定日志文件的权限。

- 使pg_upgrade创建一个脚本增量 生成更精确的优化统计(Bruce Momjian)

  这降低了升级后产生最小的集群统计所需要的时间。

- 允许pg_upgrade更新没有 `postgres` 数据库的旧的集群(Bruce Momjian)

- 允许pg_upgrade处理新的或旧的数据库丢失情况， 只要它们是空的(Bruce Momjian)

- 允许pg_upgrade处理配置目录安装(Bruce Momjian)

- 在pg_upgrade中， 添加 `-o` / `-O` 选项传递参数到服务器(Bruce Momjian)

  对配置目录安装有作用。

- 改变pg_upgrade使用缺省port 50432(Bruce Momjian)

  这有助于在更新期间避免意外的客户端连接。

- 减少pg_upgrade集群锁定(Bruce Momjian)

  具体来说， 如果使用链接模式， 那么只锁定旧的集群。 并且在存储模式之后执行。

E.8.3.13.2. pg_stat_statements

- 允许pg_stat_statements通过SQL文本标准化聚集类似查询(Peter Geoghegan, Tom Lane)

  使用非参数化SQL应用程序的用户可以没有详细的日志分析监控查询性能。

- 增加脏块和写入块计算以及读/写时间到pg_stat_statements (Robert Haas, Ants Aasma)
- 避免pg_stat_statements来自 `PREPARE` 和 `EXECUTE` 命令的重复计算(Tom Lane)

E.8.3.13.3. sepgsql

- 在全局对象上支持 `SECURITY LABEL` (KaiGai Kohei, Robert Haas)

  具体来说，添加数据库安全标签，表空间，和角色。

- 允许sepgsql接受数据库标签(KaiGai Kohei)
- 在各种对象的创建过程中执行sepgsql权限检查(KaiGai Kohei)
- 添加 `sepgsql_setcon()` 和相关函数以控制sepgsql安全域(KaiGai Kohei)
- 添加sepgsql用户空间访问缓存以提高性能(KaiGai Kohei)

E.8.3.14. 文档

- 使用网站上的样式表添加规则用以随意编译HTML文档(Magnus Hagander)

  使用 `gmake STYLE=website draft` .

- 改善 `EXPLAIN` 文档(Tom Lane)
- 记录用户/数据库名通过命令行工具如vacuumdb 使用双引号保存(Bruce Momjian)
- 记录通过客户端MD5认证返回的实际字符串(Cyan Ogilvie)
- 反对在 `CREATE TEMP TABLE` 中使用 `GLOBAL` 和 `LOCAL` (Noah Misch)

  PostgreSQL早已把这些关键词看作没有操作， 并且继续这样做；但在未来他们可能意味着SQL标准的内容， 所以应用程序应避免使用它们。

E.9. 发布9.1.10

> 发布日期: 2013-10-10

该发布包含了9.1.9的各种修复。 关于9.1主要版本新功能的信息， 请参阅Section E.19。

E.9.1. 迁移到版本9.1.10

为了运行9.1.X不需要转储/恢复。

同时，如果你是从9.1.6更早版本更新，参阅9.1.6的发布说明。

E.9.2. 变化

- 防止多字节编码中非ASCII非双引号标识符 的小写转换(Andrew Dunstan)

  以前的操作是错误的而且混乱的

- 当 `wal_level = hot_standby` 的时候， 修复后端写进程中检查点内存泄露。(Naoya Anzai)
- 修复通过 `lo_open()` 故障产生的内存泄露。(Heikki Linnakangas)
- 当 `work_mem` 正使用大于24GB的内存时， 那么修复内存过量使用错误。(Stephen Frost)
- 可串行化快照修复(Kevin Grittner, Heikki Linnakangas)
- 修复libpq SSL死锁错误(Stephen Frost)
- 修复线程libpq应用中可能的SSL网络堆变化(Nick Phillips, Stephen Frost)
- 正确计算估计布尔列包含许多NULL值的行(Andrew Gierth)

  当估计计划成本时，先前的测试像 `col IS NOT TRUE` 和 `col IS NOT FALSE` 没有合理的NULL值因素。

- 阻止叠加 `WHERE` 子句到不安 全的 `UNION/INTERSECT` 子查询中(Tom Lane)

  以前这样叠加可能产生错误。

- 修复通过不恰当地处理日 期类型修饰符产生的罕见 的 `GROUP BY` 查询错误(Tom Lane)
- 修复有删除列的外表的pg_dump (Andrew Dunstan)

  先前这种情况可能导致pg_upgrade错误。

- 重新安排相关扩展规则 的pg_dump处理和事件触发(Joe Conway)

- 如果通过 `pg_dump -t` 或者 `-n` 指定， 那么强制扩展表转储(Joe Conway)
- 允许转储编码更好地处理基本表上已删除的列(Tom Lane)
- 使用显示正确格式名的目录 归档修复 `pg_restore -l` (Fujii Masao)
- 正确记录使用 `UNIQUE` 和 `PRIMARY KEY` 语法创建的 索引注释(Andres Freund)

  这将修复并行pg_restore故障。

- 清理切换之前合理保证WAL文件传输(Fujii Masao)

  以前， 在备库上所有WAL文件被取代之前可能关闭流复制连接。

- 在恢复期间提高WAL段时间线处理(Heikki Linnakangas)
- 修复 `REINDEX TABLE` 和 `REINDEX DATABASE` 以 恰当的重新生效约束并且标记无效索引为有效(Noah Misch)

  `REINDEX INDEX` 一直正常工作。

- 在并发 `CREATE INDEX CONCURRENTLY` 操作期间修复可能死锁(Tom Lane)
- 修复 `regexp_matches()` 处理零长度匹配(Jeevan Chalke)

  先前，零长度匹配像'^'可以返回很多匹配。

- 修复过于复杂的正则表达式的错误(Heikki Linnakangas)
- 为反向引用结合非贪婪量词 修复正则表达式匹配错误(Jeevan Chalke)
- 避免 `CREATE FUNCTION` 检查 `SET` 变量除非启动函数体检查(Tom Lane)
- 允许 `ALTER DEFAULT PRIVILEGES` 在模式上操作不需要CREATE权限(Tom Lane)
- 放宽用于查询中关键字的限制(Tom Lane)

  特别地，放宽角色名称，语言名字， `EXPLAIN` 和 `COPY` 选项，以及 `SET` 值的关键字限制。 这允许 `COPY ... (FORMAT BINARY)` 事先 `BINARY` 需要单引号。

- 修复 `pgp_pub_decrypt()` 因此为带有密码的秘钥工作(Marko Kreen)
- pg_upgrade使 用 `>pg_dump --quote-all-identifiers` 避免在发布之间关键字改变的问题(Tom Lane)
- 在清理无索引表中删除少有的不正确的警告(Heikki Linnakangas)
- 在已取消文件截断请求后提高分析统计生成(Kevin Grittner)
- 当在预备查询中执行事务控制命令 (比如 `ROLLBACK`)时， 避免可能的失败(Tom Lane)
- 允许在所有平台上无穷大的各种拼写(Tom Lane)

支持无穷大的值是"inf", "+inf", "-inf", "infinity", "+infinity"和"-infinity"。

- 扩展记录和数组比较行的能力(Rafal Rzepecki,Tom Lane)
- 为了Israel, Morocco, Palestine,Paraguay中DST变化规律的DST变化规律 更新时间区域数据文件到tzdata发布2013d。 同时为Macquarie Island修正历史区域数据(Tom Lane)

E.10. 发布9.1.9

> 发布日期: 2013-04-04

该版本包含了9.1.8的各种修复。 关于9.1主要版本的新功能信息， 请参阅Section E.19。

E.10.1. 迁移到版本9.1.9

为了运行9.1.X不需要转储/恢复。

然而，该发布修正了GiST索引管理的一些错误。 在安装这个更新之后， `REINDEX` 任何GiST索引满足一个或更多个下面描述的条件是明智的。

同时，如果你正从9.1.6更早版本中更新，参阅9.1.6的发布说明。

E.10.2. 变化

- 修复服务器命令行开关安全解析(Mitsumasa Kondo, Kyotaro Horiguchi)

  包含以"`-`"开头的数据库名字的连接请求可以用来损坏或者破坏 服务器的数据目录文件，即使最终拒绝该请求。(CVE-2013-1899)

- 在每个postmaster子进程中重置OpenSSL随机状态(Marko Kreen)

  通过 `contrib/pgcrypto` 函数产生的随机数 可能对于另外一个猜测的数据库用户相对容易， 避免这种情况。 当postmaster使用 `ssl` = `on` 被配置时， 该风险是非常显著的，但是大多数连接不能使用SSL加密。(CVE-2013-1900)

- 使用REPLICATION权限检查测试当前用户不是认证用户(Noah Misch)

  一个未经授权的数据库用户可以利用 这个错误调用 `pg_start_backup()` 或者 `pg_stop_backup()` , 因此可能干扰常规备份的创建。(CVE-2013-1901)

- 当不适合这样执行时， 不使用"fuzzy"几何比较修复GiST索引。 (Alexander Korotkov)

  核心几何类型使用"fuzzy"等式执行比较， 但是 `gist_box_same` 必须执行 精确比较， 否则GiST索引使用它可能变得不一致。 安装这个更新之后， 用户应该在 `box` ， `polygon` ， `circle` 或者 `point` 列上 `REINDEX` 任何GiST索引， 因为所有这些使用 `gist_box_same` 。

- 修复不正确的范围并集以及 为了可变宽度数据类型使用 `contrib/btree_gist` 的GiST索引中惩罚逻辑， 也就是 `text` ， `bytea` ， `bit` 和 `numeric` 列(Tom Lane)

这些错误可能导致不一致索引， 其中一些出现的关键字不会被搜索发现， 并且在无用的索引膨胀中， 在安装此更新后建议用户 `REINDEX` 这种索引。

- 修复为多列索引在GiST页中分离代码的错误(Tom Lane)

  这些错误可能导致不一致索引， 其中一些出现的关键字不会被搜索发现， 并且在索引中是不必要的无效的搜索。 在安装此更新后建议用户 `REINDEX` 多列GiST索引。

- 修复 `gist_point_consistent` 处理 模糊一致性(Alexander Korotkov)

  在 `point` 列GiST索引上的索引扫描可能有时 产生不同于顺序扫描的结果， 因为 `gist_point_consistent` 不同意底层操作编码关于是否精确或者模糊的执行比较。

- 在WAL重放中修复缓冲区泄露(Heikki Linnakangas)

  在回放期间这个缺陷可能产生"不正确 的本地针数"错误，使得恢复不可能。

- 修复 `DELETE RETURNING` 中的紊乱情况(Tom Lane)

  在这样的情况下， `DELETE RETURNING` 试图 从当前进程不再有任何针的 共享缓冲区中抓取数据。 如果一些其他进程同时改变缓冲区， 这将导致垃圾 `RETURNING` 输出， 甚至崩溃。

- 修复规则表达式编译中的无限循环风险(Tom Lane, Don Porter)

- 修复规则表达式编译中潜在的空指针引用(Tom Lane)

- 合适的地方修复 `to_char()` 只 使用ASCII大小写折叠规则(Tom Lane)

  这种修复一些区域独立化的模板模式的不当行为，但是 在Turkish区域中胡乱操作" `I` "和" `i` "。

- 修复时间戳 `1999-12-31 24:00:00` 不必要的拒绝(Tom Lane)

- 当一个事务执行 `UNLISTEN` 然后 `LISTEN` 时，修复逻辑错误(Tom Lane)

  该会话根本不监听通知事件，尽管它确实在这种情况下应该监听。

- 在列被添加到依赖于其他视图的 视图中之后修复可能的规划器崩溃(Tom Lane)

- 删除无效的"picksplit不支持 的二次分裂"日志消息(Josh Hansen, Tom Lane)

  该消息似乎被添加到从未写入的期望代码中， 并且可能从来不是，因为二次分裂的GiST的缺省处理 实际上相当好。 所以停止打扰关于它的最终用户。

- 修复发送会话的 最后几个事务提交/终止计数到统计收集器的可能错误(Tom Lane)

- 消除在PL/Perl中的 `spi_prepare()` 函数的内存泄露(Alex Hunsaker, Tom Lane)

- 修复pg_dumpall以处理正确包含" `=` "的 数据库名字(Heikki Linnakangas)

- 当给定一个不正确的连接字符串时， 避免pg_dump中崩溃(Heikki Linnakangas)

- 忽略pg_dump和pg_upgrade中 的无效索引(Michael Paquier, Bruce Momjian)

  备份无效索引可能导致恢复时间的问题， 比如如果索引创建失败的原因是它试图强制 不满足表的数据的唯一性条件。同时，如果索引创建实际上仍然在进行中， 认为它是一个不受约束的DDL变化似乎是合理的，其中pg_dump 不期望备份。pg_upgrade现在也跳过无效索引而非失败。

- 在pg_basebackup中，当备份表空间时， 仅仅包含当前服务器版本的子目录 (Heikki Linnakangas)

- 在pg_basebackup和 pg_receivexlog中添加服务器版本检查， 因此它们与不工作的版本联合失败(Heikki Linnakangas)

- 修复`contrib/pg_trgm`的`similarity()` 函数为少于三个的字符串返回零(Tom Lane)

  先前它返回`NaN`由于内部除以零。

- 为了Chile, Haiti, Morocco, Paraguay和一些Russian区域中DST变化规律更 新时间区域数据文件到tzdata发布2013b。 同时为更多地方修正历史区域数据。

  同时，为俄罗斯和其他地方的最近变化更新时区缩写文件： `CHOT`，`GET`，`IRKT`，`KGT`，`KRAT`，`MAGT`，`MAWT`，`MSK`，`NOVT`，`OMST`，`TKT`，`VLAT`，`WST`，`YAKT`，`YEKT`现在遵从他们当前的含义， 以及`VOLT` (Europe/Volgograd)和`MIST`(Antarctica/Macquarie) 被添加到缺省缩写列表中。

E.11. 发布9.1.8

> 发布日期: 2013-02-07

此版本包含了9.1.7各种修复。 关于9.1主要版本的新功能的信息， 参阅Section E.19。

E.11.1. 迁移到9.1.8

为了运行9.1.X不需要转储/恢复。

然而，如果你是从早于9.1.6的版本上更新，参阅9.1.6发布说明。

E.11.2. 变化

- 防止来自SQL的`enum_recv`的执行(Tom Lane)

  该函数被错误声明， 允许简单SQL命令导致服务器崩溃。 原则上攻击者可以使用它检查服务器内存的内容。 我们该感谢Sumit Soni (通过Secunia SVCRP)报告这个问题。 (CVE-2013-0255)

- 当检测WAL回放期间达到一致性数据库状态时，修复多个问题。 (Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- 当截断关系文件时，更新最小恢复点(Heikki Linnakangas)

  一旦丢弃数据，在时间轴早一点的时候停止恢复不再安全。

- 在改变恢复目标时间轴后修复WAL段再循环(Heikki Linnakangas)

- 在热备模式中修复失去的取消(Noah Misch, Simon Riggs)

  取消热备份查询冲突的需要有时会被错过， 允许这些查询来查看不一致数据。

- 防止用户可以连接之前从暂停中恢复暂停功能(Tom Lane)

- 修复SQL语法以允许来自子SELECT结果的下标或者字段选择(Tom Lane)

- 解决繁忙工作负载中自动清理截断的性能问题(Jan Wieck)

  在表格末尾的空白页截断需要排他锁， 但自动清理编码失败（并且释放表锁） 当有冲突的锁请求时，在负载下， 很可能截断永远都不会发生，导致表的膨胀。 通过执行局部截断进行修复，释放锁，然后 试图重新获取锁并且继续。 该修复在自动清理释放锁之前冲突请求到达之后将大大 减少平均时间。

- 当扫描 `pg_tablespace` 的时候， 防止竞争条件(Stephen Frost, Tom Lane)

  如果有 `pg_tablespace` 项的并发更新， 那么 `CREATE DATABASE` 和 `DROP DATABASE` 可能行为不当。

- 防止 `DROP OWNED` 试图删除整个数据库或者表空间(Álvaro Herrera)

  为了安全起见， 这些对象的所有权必须被重新分配，而不是删除。

- 修复 `vacuum_freeze_table_age` 实现中的错误(Andres Freund)

  在安装中不只存在 `vacuum_freeze_min_age` 事务， 这个错误防止自动清理使用部分表扫描， 因此相反可能会发生全表扫描。

- 当 `RowExpr` 或者 `XmlExpr` 被解析两次时， 避免不当行为(Andres Freund, Tom Lane)

  这个错误在上下文中是用户可见的， 比如 `CREATE TABLE LIKE INCLUDING INDEXES` 。

- 提高在哈希表大小计算中防御整数溢出(Jeff Davis)

- 在服务器崩溃之后修复忽略剩余临时表错误(Tom Lane)

- 拒绝 `to_date()` 中超期范围日期(Hitoshi Harada)

- 修复 `pg_extension_config_dump()` 以适当处理扩展更新情况(Tom Lane)

  这个函数现在将取代目标表的任何已经存在项， 使它可以用于扩展更新脚本。

- 修复函数的PL/Python的处理作为多表上的触发器(Andres Freund)

- 确保非ASCII提示符字符串被翻译成Windows上正确代码页(Alexander Law, Noah Misch)

  这个错误影响psql和一些其他客户端程序。

- 当不连接数据库时， 修复psql's `\?` 命令中可能的崩溃(Meng Qingzhong)

- 当正在运行pg_basebackup时， 如果删除关系文件，那么修复可能的错误(Heikki Linnakangas)

- 当在热备份服务器上运行时，使得pg_dump 排除未记录表的数据(Magnus Hagander)

  因为数据在备用服务器上是不可用的，这可能会失败， 因此它似乎认为最方便的是自动假设 `--no-unlogged-table-data` 。

- 修复pg_upgrade以安全处理无效索引(Bruce Momjian)

- 修复libpq的 `PQprintTuples` 中一个字节缓冲溢出(Xi Wang)

  这个以往的函数 不再通过PostgreSQL自身被用于任何地方， 但是它仍然可能通过一些客户端代码被使用。

- 使得ecpglib正确使用已翻译信息(Chen Huajun)
- 在MSVC上正确安装ecpg_compat和 pgtypes库(Jiang Guiqing)
- 如果它不是通过系统提供的， 那么在libecpg中 包含`isinf()`版本(Jiang Guiqing)
- 重新配置已提供函数的配置测试， 因此它不会被libedit/libreadline假输出蒙骗。
- 确保随时间变化增加的Windows编译数(Magnus Hagander)
- 当为Windows交叉编译时， 使pgxs编译带有正确`.exe`后缀的可执行文件(Zoltan Boszormenyi)
- 添加新的时区缩写`FET` (Tom Lane)

  这用于一些东欧时区。

E.12. 发布9.1.7

> 发布日期: 2012-12-06

此版本包含了9.1.6的各种修复。 关于9.1主要版本中新功能的信息， 参阅Section E.19。

E.12.1. 迁移到版本9.1.7

为了运行在9.1.X上不需要转储/恢复。

然而，如果你正从9.1.6早期版本更新，那么参阅9.1.6发布说明。

E.12.2. 变化

- 修复与 `CREATE INDEX CONCURRENTLY` 相关的多个错误(Andres Freund, Tom Lane)

  当改变索引的 `pg_index` 行状态时， 修复 `CREATE INDEX CONCURRENTLY` 使用适当更新。 这可以防止导致并发会话错过更新目标索引的竞争条件， 因此导致崩溃同时创建索引。

  同时，修复各种其他操作以确保他们忽略来源于 失败的 `CREATE INDEX CONCURRENTLY` 命令的无效索引。 这些中最重要的是 `VACUUM` ， 因为在采取的纠正措施用于修复或删除无效索引之前， 在表上自动清理可以很容易地被运行。

- 在WAL回放期间修复缓冲区锁定(Tom Lane)

  当回放影响多页的WAL记录时，WAL回放代码对于锁定缓冲区不够小心。 这可能热备份查询瞬时看到不一致的状态， 导致错误结果或意外的失败。

- 修复GIN索引的WAL生成逻辑的错误(Tom Lane)

  如果发生残缺页故障，那么这会导致索引崩溃。

- 当推进热备份服务器正常运行时， 正确删除启动进程的虚拟XID锁(Simon Riggs)

  这种监督可以防止 某个操作的后续执行比如 `CREATE INDEX CONCURRENTLY` 。

- 在待机模式下 避免虚假"out-of-sequence timeline ID"错误(Heikki Linnakangas)

- 在接收到停机信号后，防止发起新的子进程的postmaster(Tom Lane)

  这个错误可能导致关闭更长比它应该的， 或者没有额外用户操作甚至不能完成。

- 当内存不足时，那么避免内部哈希表的崩溃(Hitoshi Harada)

- 避免已删除表的文件描述符可以保持打开以往事务结束(Tom Lane)

这可以减少长时间已删除表继续占用磁盘空间的问题。

- 当一个新的子进程不能为它的锁创建一个管道，那么防止 数据库端崩溃以及重启(Tom Lane)

  尽管新的进程失败， 没有充分的理由强迫数据库端重启，所以避免它。 当内核基本没有文件描述符时，那么这提高了鲁棒性。

- 修复外连接上非严格等价分句规划(Tom Lane)

  规划器可以从等同于其它东西的非严格建构的分句中获取不正确约束， 比如 当 `foo` 来源于外连接失效端时， `WHERE COALESCE(foo, 0) = 0` 。

- 在继承树上使用索引优化的 `MIN` / `MAX` 修复 `SELECT DISTINCT` (Tom Lane)

  该规划器伴随着"没有重新找到MinMaxAggInfo记录"给定这些结合因素而失败。

- 提高从等价类证实排他约束的规划器能力(Tom Lane)

- 修复在哈希子计划中部分行匹配以正确处理交叉类型情况(Tom Lane)

  这影响多列 `NOT IN` 子计划， 比如 `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` 当比如 `b` 和 `y` 分别是 `int4` 和 `int8` 的时候。这个错误导致错误结果 或者取决于依赖于涉及到的特定数据类型的崩溃。

- 当为 `AFTER ROW UPDATE/DELETE` 触发器重新读取旧的元组时， 那么获取缓冲锁(Andres Freund)

  在非常罕见的情况下， 这一疏忽可能导致传递不正确数据到触发器 `WHEN` 条件， 或者为外键执行触发器预检查逻辑。这可能导致崩溃， 或者关于是否触发触发器的错误决定。

- 修复 `ALTER COLUMN TYPE` 以 正确处理已继承的检查约束(Pavan Deolasee)

  这在8.4版本之前正常运行， 并且现在在8.4以及以后也正常运行。

- 修复 `ALTER EXTENSION SET SCHEMA` 的错误 以移动一些子对象到新的模式(Álvaro Herrera, Dimitri Fontaine)

- 修复 `REASSIGN OWNED` 以 处理表空间授权(Álvaro Herrera)

- 忽略视图系统列中不正确的 `pg_attribute` 项(Tom Lane)

  视图没有任何系统列。然而，当转换表到视图时，我们忘了删除这项。 9.3以及以后的正确修复，但是在以前的分支中我们需要防御已经存在的 错误转换视图。

- 修复规则输出以正确备份 `INSERT INTO` `_table_` DEFAULT VALUES(Tom Lane)

- 当在查询中有很多 `UNION` / `INTERSECT` / `EXCEPT` 子句时， 避免栈溢出(Tom Lane)

- 当将最小可能整数值除以-1时，避免平台相关故障(Xi Wang, Tom Lane)

- 修复日期解析中字符串可能的访问先前终止部分(Hitoshi Harada)
- 在检查点期间如果产生XID重叠，修复先前XID纪元错误，并且 `wal_level` 是 `hot_standby` (Tom Lane, Andres Freund)

  当这个错误对PostgreSQL自身没有特别影响， 对于依赖于 `txid_current()` 和相关函数：TXID值可能出现回退， 这是一个不好的应用。

- 修复页面边界上 `pg_stat_replication`. `sync_state` 的显示(Kyotaro Horiguchi)
- 如果为了Unix域套接字路径名长度超过了特定平台限制， 那么产生一个可理解的错误信息(Tom Lane, Andrew Dunstan)

  以前，这可能导致一些无用的东西，比如 "域名解析中不可恢复错误"

- 当发送复合列值到客户端时，修复内存泄露(Tom Lane)
- 使得pg_ctl对读取 `postmaster.pid` 文件更具有鲁棒性(Heikki Linnakangas)

  修复竞争条件和可能的文件描述符泄露。

- 如果提出错误的编码数据，修复psql中可能的错误， 并且 `client_encoding` 设置是客户端编码，比如SJIS (Jiang Guiqing)
- 在不是归档的预先数据部分的数据中使得 pg_dump备份 `SEQUENCE SET` 项(Tom Lane)

  该变化修复被标记为扩展配置表的序列的备份。

- 修复在 `tar` 输出格式中通过pg_dump 发出的 `restore.sql` 脚本中的错误(Tom Lane)

  该脚本可能在它的名字包含大写字符的表中完全失败。 另外，使得脚本有能力存储数据到 `--inserts` 模式和规则的COPY模式。

- 修复pg_restore以 接受POSIX标准 `tar` 文件(Brian Weaver, Tom Lane)

  pg_dump的 `tar` 输出模式的原编码产生与 POSIX标准不完全一致的文件。这是9.3版本的校正。 这个补丁更新先前分支，以致于它们接受正确的和不正确格式， 当发布9.3时，希望避免兼容性问题。

- 修复通过pg_basebackup到POSIX一致 发出的 `tar` 文件(Brian Weaver, Tom Lane)
- 当给定一个数据目录的相对路径时，修复pg_resetxlog 以正确定位 `postmaster.pid` (Tom Lane)

  这个错误可能导致pg_resetxlog没有注意到 有使用数据目录的活跃postmaster。

- 修复libpq的 `lo_import()` 和 `lo_export()` 函数以正确报告文件I/O错误(Tom Lane)
- 修复嵌套结构指针变量的ecpg处理(Muhammad Usama)

- 修复ecpg的 `ecpg_get_data` 函数 以正确处理数组(Michael Meskes)
- 当正在检查页的时候，使得 `contrib/pageinspect` 的btree 页检查函数带有缓冲锁(Tom Lane)
- 确保 `make install` 为扩展创建 `extension` 安装目录(Cédric Villemain)

  先前，如果在扩展的Makefile中设置 `MODULEDIR`，可以忽略这步。
- 修复pgxs在AIX上支持创建可加载模块(Tom Lane)

  编译起初源码树外部模块在AIX上不工作。
- 在Cuba, Israel, Jordan, Libya, Palestine, Western Samoa以及Brazil地区中为DST变化规律更新时区数据文件到 tzdata发布2012j。

E.13. 发布9.1.6

> 发布日期: 2012-09-24

该发布中包含来自9.1.5的各种修复。 关于9.1主要版本的新功能的信息， 参阅Section E.19。

E.13.1. 迁移到版本9.1.6

为了运行9.1.X不需要转储/恢复。

然而，你可能需要执行 `REINDEX` 从下面第一个日志项描述的数据损坏 错误的影响中恢复。

另外，如果你从9.1.4更早版本更新， 查看9.1.4的发布说明。

E.13.2. 变化

- 在WAL回放期间修复共享缓冲区的持久性标记(Jeff Davis)

  这个错误会导致缓冲区在检查点期间不被写出来， 如果服务器没有写入缓冲区崩溃后，导致数据损坏。 在任何服务器崩溃恢复之后发生崩溃， 但它显著的可能发生在备用子服务器上， 因为这些执行更多WAL回放。 有btree和GIN索引损坏的低概率。 有表"可见视图"损坏的更高概率。 幸运的是，可见视图是9.1中非关键数据， 因此9.1安装中这样的损坏最糟糕的后果是清理短暂无效。 表正确的数据无法被这个错误损坏。

  虽然没有索引损坏，由于这个错误已经在该字段发生， 作为预防措施建议在更新到 9.1.6 之后在方便的时候产品安装 `REINDEX` 所有btree和GIN索引。

  同时，如果你打算做适当升级到9.2.X， 在做这些之前所以建议执行所有表的 `VACUUM`， 当 `vacuum_freeze_table_age` 设置为零时，这将确保在9.2.X可以依赖它之前校正可见视图中的任何残留的错误数据。 可以调整 `vacuum_cost_delay` 以减少 清理的性能影响，而造成它需要更长时间完成。

- 修复执行器参数的规划器分配， 并且为CTE规划节点修复执行器的重新扫描逻辑(Tom Lane)

  这些错误可以导致来自 扫描同一 `WITH` 子查询多次的查询的错误结果。

- 当 `default_transaction_isolation` 设置为 `serializable` 的时候， 修复错误操作 (Kevin Grittner, Tom Lane, Heikki Linnakangas)

  症状包含Windows启动过程的死机， 以及热备操作的死机情况。

- 提高涉及前缀的文本搜索查询的选择行估计， 比如 `_word_``:*` 模式(Tom Lane)

- 提高GiST索引中页分裂决定 (Alexander Korotkov, Robert Haas, Tom Lane)

  多列GiST索引由于这个错误可能遭受意外膨胀。

- 如果仍然持有特权，那么修复终止的级联权限撤销(Tom Lane)

  如果我们撤销一些角色 `_X_` 的grant选项，但是 `_X_` 仍然认为该选项通过其他人的grant。 我们不应该递归地撤销 `_X_` 授予的角色 `_Y_` 的 相应特权。

- 不接受包含分配给它们的模式的扩展(Thom Brown)

  这种情况创建了困惑pg_dump和 其他一些事情的循环依赖。 它也令人困惑，因此不接受它。

- 提高热备份不当配置错误的错误信息(Gurjeet Singh)

- 尝试configure探查 `mbstowcs_l` (Tom Lane)

  修复AIX一些版本上的编译错误。

- 当使用PL/Perl时，修复 `SIGFPE` 的处理(Andres Freund)

  Perl重置进程的 `SIGFPE` 处理器到 `SIG_IGN` ， 这可能在以后导致崩溃。 在初始化PL/Perl之后恢复正常Postgres信号处理程序。

- 当被执行时，如果重新定义递归的PL/Perl函数， 则防止PL/Perl崩溃(Tom Lane)

- 解决PL/Perl中可能的错误优化(Tom Lane)

  一些Linux发布包含 导致PL/Perl中不正确编译代码的 `pthread.h` 不正确版本， 如果PL/Perl函数调用抛出错误的另外一个，那么导致崩溃。

- 修复 `contrib/pg_trgm` 's `LIKE` 模式 分析代码中的错误(Fujii Masao)

  如果模式包含 `LIKE` 转义字符， 那么使用三线性索引的 `LIKE` 查询可以产生错误结果。

- 修复Windows上行尾的pg_upgrade的处理(Andrew Dunstan)

  以前，pg_upgrade可能添加或者删除运输返回比如函数体的地方。

- Windows上，使得pg_upgrade在它 发出的脚本中使用反斜杠路径分隔符(Andrew Dunstan)

- 删除来自 pg_upgrade的pg_config的不必要依赖(Peter Eisentraut)

- 为了Fiji中的DST变化规律更新时区数据文件到tzdata发布2012f。

E.14. 发布9.1.5

> 发布日期: 2012-08-17

这个版本包含9.1.4的各种修复。关于9.1主要版本的新功能的信息， 参阅Section E.19。

E.14.1. 迁移到版本9.1.5

为运行9.1.X不需要转储/恢复。

然而，如果你从9.1.4的更早版本更新， 参阅9.1.4的发布说明。

E.14.2. 变化

- 防止通过XML实体引用访问外部文件夹/URL(Noah Misch, Tom Lane)

  `xml_parse()` 可以尝试读取外部文件夹或者URL作为需要解决DTD 以及XML值中的实体引用，从而允许未授权数据库用户尝试读取与数据库服务器权限的数据。 当外部数据还没直接返回给用户时，如果数据没有解析为有效XML，那么它的一部分可能会暴露 在错误信息中；并且无论如何检查文件是否存在的能力可能对攻击者有用。

- 阻止通过 `contrib/xml2` 的 `xslt_process()` 访问外部文件夹/URL(Peter Eisentraut)

  libxslt提供通过样式表命令读写文件夹和URL的能力， 从而允许未授权数据库用户以读写带有数据库服务器权限的数据。 禁止通过libxslt的安全选项的正确使用。(CVE-2012-3488)

  同时，删除 `xslt_process()` 的能力从外部文件夹/URL中读取文件和样式表。 当这是已证明"特性"时，那么它被长期视为坏主意。 为了CVE-2012-3489修复打破该能力，而不是付出努力尝试修复它， 我们只是打算简单地删除它。

- 防止btree索引页过早回收利用(Noah Misch)

  当我们允许只读事务略过已分配XID时， 当只读事务仍然运行到它时， 那么我们介绍已删除btree页可以被重新利用的可能性。 这会导致不正确索引搜索结果。 在该字段产生错误的可能性很低，因为时间要求， 但尽管如此它应该被修复。

- 修复带有新创造或者重新设置序列的碰撞安全漏洞(Tom Lane)

  如果在新创造或者重新设置序列上执行 `ALTER SEQUENCE` ， 然后在它上精确执行一个 `nextval()` 调用， 然后服务器崩溃了，WAL回放可以恢复序列到似乎没有执行 `nextval()` 的状态下， 然而允许通过下一个 `nextval()` 调用再次返回第一个序列值。

特别是这可以表现为 `serial` 列，因为串行列的序列的创建包含 `ALTER SEQUENCE OWNED BY` 步骤。

- 修复 `enum` 类型值比较的竞争条件 (Robert Haas, Tom Lane)

  当遇到一个被添加的枚举值时， 比较可能失败， 因为当前查询开始。

- 当不再热备份时， 修复 `txid_current()` 以报告正确时代(Heikki Linnakangas)

  这修复了上一个次要版本介绍的回归。

- 防止不当的复制连接选择作为同步备用(Fujii Masao)

  主库可能不恰当选择虚假服务器 比如pg_receivexlog或者pg_basebackup 作为同步备用，然后无限期等待它们。

- 当主事务有很多子事务时， 修复热备启用错误(Andres Freund)

  这个错误导致故障报告 为"无效XID插入到KnownAssignedXids中"。

- 在 `pg_start_backup()` 之后确保 `backup_label` 文件是fsync (Dave Kerr)

- 在walsender进程中修复超时处理(Tom Lane)

  WAL发送后端进程以建立SIGALRM处理程序， 意味着它们会永远等待超时发生的一些情况。

- 在每个后端通过walwriter冲洗后意识到walsender (Andres Freund, Simon Riggs)

  当工作负载只包含异步提交事务时， 这大大减少了复制延迟。

- 修复 `LISTEN` / `NOTIFY` 更好地处理I/O问题， 比如磁盘空间不足(Tom Lane)

  在写入失败后，尝试发送更多 `NOTIFY` 消息的所有子序列可能 带有信息如"不能从文件"pg*notify/`_nnnn* `" 偏移量` *nnnnn*`: 成功读取"而失败。

- 仅仅允许autovacuum通过直接的封锁进程被自动取消(Tom Lane)

  原代码可能允许某些情况下的不一致操作； 特别是，在少于 `deadlock_timeout` 宽限期后可以取消autovacuum。

- 改善autovacuum取消记录(Robert Haas)

- 修复日志收集器以致于在服务器启动后 第一个日志循环期间运行 `log_truncate_on_rotation` (Tom Lane)

- 修复 `WITH` 附属于嵌套设置操作 (`UNION` / `INTERSECT` / `EXCEPT`) (Tom Lane)

- 确保参照子查询的整行不包含任何额外的 `GROUP BY` 或者 `ORDER BY` 列(Tom Lane)

- 在 `ALTER TABLE ... ADD CONSTRAINT USING INDEX` 期间修复产生的依赖(Tom Lane)

这个命令为索引留下了多余的 `pg_depend` 项， 这可能混淆后期操作， 尤其是索引列之一上的 `ALTER TABLE ... ALTER COLUMN TYPE` 。

- 修复 `REASSIGN OWNED` 以影响扩展(Alvaro Herrera)
- 在 `CREATE TABLE` 期间 `CHECK` 约束和索引定义中不 允许拷贝整行引用(Tom Lane)

  这种情况可以产生带有 `LIKE` 或者 `INHERITS` 的 `CREATE TABLE` 。 复制整列变量被错误地标记带有不是一个新的原来表的行类型。 为 `LIKE` 拒绝理由似乎是合理的， 因为行类型可能后面会分散。 为 `INHERITS` 我们理论上应该接受它， 伴随对父表的行类型的隐含 胁迫；但比起后端补丁似乎是安全的需要更多的工作。

- 修复 `ARRAY(SELECT ...)` 子查询中的内存泄露 (Heikki Linnakangas, Tom Lane)
- 修复规划器传递正确规则排序到操作符选择性估计者(Tom Lane)

  任何核心选择性估计函数先前不需要这个， 但是第三方代码可能需要它。

- 从正则表达式修复常见前缀提取(Tom Lane)

  该代码被量化的括号子表达式搞糊涂了，比如 `^(foo)?bar` 。 这将导致这种模式不正确的搜索索引优化。

- 修复 `interval` 常量中带有分析符号 `_hh_``:``_mm_` 和 `_hh_``:``_mm_``:``_ss_` 字段的错误(Amit Kapila, Tom Lane)
- 修复pg_dump以 更好处理包含部分 `GROUP BY` 列表 的视图(Tom Lane)

  视图在 `GROUP BY` 中只列出一个主键列， 但如果他们使用其他表列进行分组，那么根据主键进行标记。 pg_dump中这样的主键依赖的恰当处理导致差的有序转储， 这充其量是效率很低的恢复而且 在最坏的情况 可能会导致一个平行的pg_restore运行的彻底失败。

- 在PL/Perl中，当使用SQL_ASCII编码时， 避免设置UTF8标记(Alex Hunsaker, Kyotaro Horiguchi, Alvaro Herrera)
- 当转换Python Unicode字符串到PL/Python中的服务器编码时， 使用Postgres的编码转换函数，而不是Python的。

  这避免了一些拐角情况问题， 值得注意的是Python不支持所有Postgres编码。 一个显著功能变化是如果服务器编码是SQL_ASCII， 你会 得到字符串UTF-8表示形式；以前， 字符串中任何非ASCII字符可以导致错误。

- 修复PostgreSQL编码映射到PL/Python中的Python编码(Jan Urbanski)
- 适当报告 `contrib/xml2` 的 `xslt_process()` 中的错误(Tom Lane)
- 为Morocco和Tokelau中DST变化规律更新时区 数据文件到tzdata发布2012e

E.15. 发布9.1.4

> 发布日期: 2012-06-04

这个发布包含来自9.1.3的各种修复。 关于9.1主要版本的新功能信息， 参阅Section E.19。

E.15.1. 迁移到版本9.1.4

为了运行9.1.X不需要转储/恢复。

然而，如果你使用 `citext` 数据类型， 并且通过运行pg_upgrade升级原先主版本，你 应该运行 `CREATE EXTENSION citext FROM unpackaged` 为了避免 `citext` 操作中排序相关故障。 如果你从包含 `citext` 数据类型 的实例中的9.1之前数据库中恢复备份，那么同样是必须的。 在更新到9.1.4之前如果你已经运行 `CREATE EXTENSION` 命令， 你反而需要手动目录更新正如下面第三个日志项解释的。

另外，如果你从9.1.2更早版本进行更新， 参阅9.1.2发布说明。

E.15.2. 变化

- 修复 `contrib/pgcrypto` 的DES `crypt()` 函数中 不正确密码转换(Solar Designer)

  如果密码字符串包含字节值 `0x80` ，那么忽略剩余的密码， 导致密码比它出现的更加弱。 使用这个修复，剩余字符串被恰当地包含在DES哈希中。 受这个错误影响的任何存储密码值将不再匹配，因此 存储值可能需要被更新。(CVE-2012-2143)

- 为一个程序语言的调用处理程序忽略 `SECURITY DEFINER` 和 `SET` 属性(Tom Lane)

  应用这些属性到调用处理器可以使服务器崩溃(CVE-2012-2655)

- 尝试 `contrib/citext` 的更新脚本修复 `citext` 数组的排序规则和超过 `citext` 的域(Tom Lane)

  发布9.1.2为 `citext` 列的排序规则和来自9.1之前安装的数据库更新或者重载中的 索引提供修复。但是这个修复是不完整的：它忽略处理数组和 `citext` 上的域。 这个发布扩展模块的更新脚本以处理这些情况。 如以前，如果你已经运行更新脚本，你将需要手动运行排序规则更新命令。 参阅9.1.2发布说明获取更多关于做这个的详细信息。

- 允许 `timestamp` 输入 中数值时区偏移量远离UTC达到16小时(Tom Lane)

  一些历史时区有大于15小时的偏移量，先前限制。 这可能导致备份数据值在重载期间被拒绝。

- 当给定时间恰恰是当前时区的最后DST转变时间时， 修复时间戳转换处理(Tom Lane)

  这次疏忽已有很长时间，但是以前没有被注意到， 因为假设大多数DST时区有未来DST转换的不明确的序列。

- 修复 `text` 到 `name` 并且 `char` 到 `name` 投射以便在 多字节编码中正确执行字符串截断(Karl Schnaitter)

- 修复 `to_tsquery()` 中内存拷贝错误(Heikki Linnakangas)

- 当在热备中执行时， 确保 `txid_current()` 报告正确时期(Simon Riggs)

- 修复子查询内PlaceHolderVars外的规划器处理 (Tom Lane)

  这个错误涉及到子SELECT，它引用来自周围查询的外部连接的空侧的变量。 在9.1中，这个错误影响的查询可能 伴随有"错误：在不被预期的地方发现上层PlaceHolderVar"而失败。 但是在9.0和8.4中，你可能默默地获得可能的错误结果，因为当需要时， 传递到子查询中的值不能定位到空。

- 修复有不是简单变量输出列的 `UNION ALL` 子查询计划(Tom Lane)

  这种情况下的规划在9.1中有着明显恶化，作为 错误修正"MergeAppend子目标列不匹配MergeAppend"错误的结果。 恢复那个修复并且以另一种方式执行它。

- 当 `pg_attribute` 非常大时， 修复缓慢会话启动(Tom Lane)

  如果 `pg_attribute` 超过了 `shared_buffers` 的四分之一， 在会话开始时有时需要缓存重建代码可以触发同步扫描逻辑， 导致它采取比正常更长的时间。 如果许多新会话马上开始，那么问题是相当严重的。

- 确保顺序扫描合理地检查查询取消(Merlin Moncure)

  遇到许多包含非活跃元组连续页的扫描不会同时响应中断。

- 确保返回之前 `PGSemaphoreLock()` 清除 `ImmediateInterruptOK` 的Windows实现(Tom Lane)

  这种疏忽意味着在同一个查询中后来收到的查询取消中断可能在不安全时间 被接受，伴随着不可预知的但不好的结果。

- 当输出视图或者规则时，安全显示整行变量(Abbas Butt, Tom Lane)

  涉及歧义名字(也就是说，该名字可以是一个表或者查询的列名)的情况被以模糊方式输出， 冒险转储和重载之后不同地解释视图或者规则。 通过附加无操作计算避免模凌两可的情况。

- 修复 `COPY FROM` 以正确处理与无效编码一致的空标记字符串(Tom Lane)

  一个空标记字符串比如 `E'\\0'` 应该工作， 并且工作于过去，但是这种情况在8.4中被打破。

- 修复 `EXPLAIN VERBOSE` 为可写CTE包含 `RETURNING` 子句 (Tom Lane)
- 在咨询锁存在下修复 `PREPARE TRANSACTION` 以正常工作(Tom Lane)

  从历史看， `PREPARE TRANSACTION` 简单忽略了任何会话持有的会话级别 咨询锁，但是这种情况在9.1中被意外损坏。

- 修复未记录表的截断(Robert Haas)
- 在 `search_path` 的非交互式分配中忽略缺失模式(Tom Lane)

  这重新排列带有旧分支的9.1的操作。先前9.1可能 为从某地比如 `ALTER DATABASE SET` 中获得的 `search_path` 设置中提及的不存在的模式而抛出错误。

- 修复用于扩展脚本的临时或者短暂表的错误(Tom Lane)

  这个包含比如在扩展更新脚本中重写 `ALTER TABLE` 的情况， 因为在该场景后使用临时表。

- 确保autovacuum工作进程恰当执行堆栈深度检查(Heikki Linnakangas)

  先前，通过自动 `ANALYZE` 调用的无限递归函数可以使工作进程崩溃。

- 修复日志收集器在高负载下没有丢失日志一致性(Andrew Dunstan)

  如果它太忙，那么收集器先前可能重新收集大的信息失败。

- 修复日志收集器以确保它在接收SIGHUP之后 重启文件旋转(Tom Lane)
- 修复GiST索引中"太多LWLocks采取"错误 (Heikki Linnakangas)
- 如果索引随后被删除，那么修复GIN索引WAL重放逻辑而不失败(Tom Lane)
- 正确检测崩溃后预备事务的SSI冲突(Dan Ports)
- 当提交一个仅仅修改临时表的事务时，避免同步复制延迟(Heikki Linnakangas)

  在这种情况下事务的提交记录不需要冲刷到备用服务器，但是 一些代码并不知道等待它发生。

- 修复pg_basebackup中的错误处理(Thomas Ogrisegg, Fujii Masao)
- 如果连接中断， 那么修复walsender不进入一个繁忙循环中(Fujii Masao)
- 修复PL/pgSQL的 `RETURN NEXT` 命令中的内存泄露 (Joe Conway)
- 当目标是函数的第一个变量时，那么修复PL/pgSQL的 `GET DIAGNOSTICS` 命令(Tom Lane)
- 确保PL/Perl包具有 `_TD` 变量(Alex Hunsaker)

  当它们被嵌套在改变当前包的函数调用中时，这个错误导致 触发器调用失败。

- 修复返回复合类型的PL/Python函数以接受结果值的字符串(Jan Urbanski)

  这种情况通过9.1附加到 允许提供其他格式的复合结果值而被意外的打断，比如词典。

- 在psql的可扩展显示(`\x`)模式中 修复内存结尾潜在访问(Peter Eisentraut)
- 当数据库包含许多对象时，那么修复pg_dump中的 一些性能问题(Jeff Janes, Tom Lane)

  如果数据库包含许多视图，或者如果许多对象在依赖循环中， 或者如果有许多拥有的序列，那么pg_dump可能会很慢。

- 当读取目录格式归档时，修复内存和pg_restore中的 文件描述符泄露(Peter Eisentraut)
- 为数据库存储在 集群中的缺省表空间中包含表的非默认表空间的情况 而修复pg_upgrade(Bruce Momjian)
- 在ecpg中，修复罕见内存泄露并且 `sqlca_t` 结构后可能覆盖一个字节(Peter Eisentraut)
- 修复 `contrib/dblink` 的 `dblink_exec()` 不泄露临时数据库连接错误(Tom Lane)
- 修复 `contrib/dblink` 以 报告错误消息中正确连接名(Kyotaro Horiguchi)
- 当删除多个大对象的时候， 修复 `contrib/vacuumlo` 以使用多个事务(Tim Lewis, Robert Haas, Tom Lane)

  当多个对象需要被删除的时候， 这个改变避免超过 `max_locks_per_transaction` 。 该操作可以被调整为具有新的 `-l` (限制)选项。

- 为了在Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria和 Tokelau Islands中DST变化规律 更新时区数据文件到tzdata发布2012c； 同时为Canada历史修正。

E.16. 发布9.1.3

> 发布日期: 2012-02-27

该发布包含了9.1.2的各种修复。 为了9.1主要版本的新功能的更多信息， 参阅Section E.19。

E.16.1. 迁移到版本9.1.3

为运行9.1.X不需要转储/恢复。

然而，如果你从9.1.2的更早版本更新， 请参阅9.1.2的发布说明。

E.16.2. 变化

- 需要为 `CREATE TRIGGER` 触发器函数上的执行权限(Robert Haas)

  这个丢失检查可能允许其他用户执行带有伪造输入数据的触发器函数， 通过安装它到它拥有的表上。 对于触发器函数标记 `SECURITY DEFINER` 是唯一重要的， 因为否则触发器函数运行为表所有者。(CVE-2012-0866)

- 删除SSL证书中常见名称长度的任意限制(Heikki Linnakangas)

  libpq和服务器截断从32字节SSL证书中提取的通用名。 这通常会导致没有什么比一个意想不到的验证失败更糟糕的，但是有一些 令人难以置信的情况，它可能允许一个证书持有者模仿另外一个。 该受害者必须有32字节长的通用名。并且攻击者必须说服可信任CA发布证书， 其中通用名有字符串作为前缀。 伪装服务器也需要一些额外的开发重定向客户端连接。(CVE-2012-0867)

- 在名字写入pg_dump说明中转换新行到空白(Robert Haas)

  pg_dump关于审查输出脚本中SQL注释发出的对象名是不谨慎的。 包含换行符的名字至少使得脚本语法上不正确。 当脚本被重新加载时，恶意制作对象名可能出现SQL注射风险。(CVE-2012-0868)

- 修复来自并行清理插入的btree索引崩溃(Tom Lane)

  通过插入引起的索引页分裂有时可以导致同时运行 `VACUUM` 而错过删除本应该删除的索引项。相应表行被删除后， 该悬挂索引项可能导致错误（比如"不能读取文件中块N..."）或者更糟， 无关行后默默的错误查询结果被重新插入到当前自由表位置。 这个错误自从发布8.2就出现了， 但是发生如此罕见以致它没有被诊断直到现在。 如果你有理由怀疑它已经在你的数据库中发生， 那么重新索引受影响索引将修复这问题。

- 修复WAL回放期间共享缓冲区临时调零(Tom Lane)

  重放逻辑有时归零并且重填共享缓冲区，因此内容瞬时无效。 在热备模式中这可以导致正在并行执行的查询看到垃圾数据。 可能导致不同症状， 但是最常见的是"无效内存分配请求大小"。

- 修复 `READ COMMITTED` 复查中数据修改 `WITH` 子计划处理(Tom Lane)

  如果父 `UPDATE` 或者 `DELETE` 命令需要被重新评估一行或多行， 由于 `READ COMMITTED` 模式中并发更新，那么包含 `INSERT` / `UPDATE` / `DELETE` 的 `WITH` 子句可能崩溃。

- 修复SSI事务清理中困境情况(Dan Ports)

  当结束一个读写串行化事务时，如果所有剩余的活跃的可串行化事务是 只读的，那么可能产生崩溃。

- 在热备份崩溃后修复postmaster以尝试重启(Tom Lane)

  当在热备模式中进行操作时如果任何后端进程崩溃，那么 逻辑错误导致postmaster终止，而不是尝试重启集群。

- 修复通过最新更新行拥有的toast值的 `CLUSTER` / `VACUUM FULL` 的处理(Tom Lane)

  这种疏忽可能导致"重复关键值违背唯一约束"错误 被报告给这些命令之一中的 toast表的索引。

- 当改变表所有者时，更新每列权限，不仅仅每个表权限(Tom Lane)

  不这样做就意味着 任何先前已授权列权限 仍然显示为已被旧的所有者授权。 这意味着既不是新所有者也不是 超级用户可以撤销目前难以寻找的到表所有者权限。

- 支持外数据封装和 `REASSIGN OWNED` 中的外服务器(Alvaro Herrera)

  如果它需要改变任何对象的所有权， 那么该命令伴随"意外数字"错误失败。

- 允许 `ALTER USER/DATABASE SET` 中 一些设置的不存在值(Heikki Linnakangas)

  允许 `default_text_search_config` ， `default_tablespace` 和 `temp_tablespaces` 被设置为不知道的名字。 这是因为它们可能在另一个数据库中已知， 该设置打算使用的地方， 或者为了表空间情况因为表空间可能不会被创建。 同样问题是先前已确认为 `search_path` ， 并且这些设置像那一个。

- 修复 `INSERT` 表达式中通过 `COLLATE` 产生的 "不支持节点类型"错误(Tom Lane)

- 当我们删除后提交表文件有问题时，避免崩溃(Tom Lane)

  删除表导致事务提交之后删除底层磁盘文件。 在失败的情况中（比如，由于错误文件权限） 那么该代码应该发出警告信息并且继续， 因为它太晚了而终止了事务。 这个逻辑已作为发布8.4被打破， 导致这种情况引起PANIC和不可重新启动的数据库。

- 在 `DROP TABLESPACE` 的WAL回放期间从发生的错误中恢复(Tom Lane)

  重播将尝试删除该表空间目录，但是 可能会失败的原因是多方面的（例如，这些目录上不正确的 所有权和权限）。以前该重播代码会恐慌， 导致数据库没有手册干预不能重新启动。 似乎记录问题并且继续是更好的，因为 删除该目录失败的唯一结果是一些浪费的磁盘空间。

- 修复为热备在记录AccessExclusiveLocks中的竞争条件(Simon Riggs)

  有时锁可能被记录为通过"事务零"持有。 这对于从服务器上产生断言失败至少是已知的， 并且可能是更严重问题的原因。

- 在WAL回放期间正确跟踪OID计数器，即使当它包围周围(Tom Lane)

  先前OID计数器可以保持在较高的值上直到系统退出回放模式。 实际结果通常为零，但是存在这样一种情况， 备用服务器提升到主服务器可能需要很长时间 增加OID计数器到一个合理值，一旦该值是必须的。

- 在崩溃恢复开始时阻止发出误导性的 "一致的恢复状态到达" 日志信息(Heikki Linnakangas)

- 修复 `pg_stat_replication`.`replay_location` 的初始值(Fujii Masao)

  以前，显示的值可能是错误的直到至少一个WAL记录已被回放。

- 修复带有 `*` 附属的正则表达式逆向引用(Tom Lane)

  而不是执行一个确切的字符串匹配， 该代码有效地接受任何满足模式子表达式引用逆向引用符号的字符串。

  类似问题仍然困扰着被嵌入到大的量化表达式中的逆向引用，而不是量词的直接主题。 这将在未来PostgreSQL发布中得以解决。

- 修复 `inet`/`cidr` 值处理中最新引进的内存泄露(Heikki Linnakangas)

  PostgreSQL的2011年12月份发布的补丁 导致了这些操作中内存泄露，这可能是重要情况比如在这样的列上 编译btree索引。

- 修复规划器的能力通过 `UNION ALL` 推动索引表达式限制(Tom Lane)

  优化这种类型通过9.1.2中另一个问题修复被无意识禁用。

- 修复引用继承表上 `UPDATE`/`DELETE` 中的 `WITH` 子句规划(Tom Lane)

  这个错误导致"不能找到CTE规划"错误。

- 修复GIN估计成本以处理 `column IN (...)` 索引条件(Marti Raudsepp)

  如果这一条件与GIN索引一起使用，那么这种疏忽通常导致崩溃。

- 当退出打开的失败事务会话，防止断言失败(Tom Lane)

  这个错误对正常编译未启用断言没有影响。

- 修复SQL语言函数中 `CREATE TABLE AS` / `SELECT INTO` 之后的悬挂指针(Tom Lane)

  在大多数情况中这导致断言启用编译中断言失败， 但是更糟结果是可能的。

- 避免Windows上syslogger中文件句柄的双关闭(MauMau)

  通常这个错误是无形的，但是当在Windows的调试版本上运行时， 它可能导致异常。

- 修复plpgsql中I/O转换关系内存泄露(Andres Freund, Jan Urbanski, Tom Lane)

  某些操作可能泄露内存直到当前函数结束。

- 解决perl的SvPVutf8()函数中的错误(Andrew Dunstan)

  当操作的typeglob或者某个只读对象比如 `$^V` 时， 该函数崩溃。 使得plperl避免传递这些给它。

- 在pg_dump中，如果扩展本身没有被备份， 那么不备份扩展的配置表的内容(Tom Lane)

- 提升继承表列的pg_dump的处理(Tom Lane)

  pg_dump处理不当的情况下，一个子列 比它的父列有不同的缺省表达式。 如果缺省文本上与父类的缺省是相同的，但不是真的 相同（例如，由于模式搜索路径的差异）不会被认为是不同的， 所以在转储和恢复后 子类可以被允许继承父的缺省。在它们的父类不能微妙地错误地被恢复的地方 子列 `非空` 。

- 为了INSERT形式表数据修复pg_restore的 直接到数据库模式(Tom Lane)

  当使用发布日期2011年九月或者十二月的pg_restore的时候， 从归档文件中恢复直接到数据库伴随 `--inserts` 或者 `--column-inserts` 选项而失败， 作为另外一个问题修复的疏忽结果。 归档文件本身没有错，而且文本模式输出是好的。

- pg_upgrade处理plpython 的共享库的重命名(Bruce Momjian)

  更新包含plpython的9.1之前数据库可能失败，因为这种疏忽。

- 允许pg_upgrade处理包含 `regclass` 列的表(Bruce Momjian)

  因为pg_upgrade现在负责保存 `pg_class` OID， 这一限制没有任何理由。

- 当正在寻找SSL客户端证书文件时，使得 libpq忽略 `ENOTDIR` 错误(Magnus Hagander)

  这允许建立SSL连接，虽然没有证书， 即使当用户的家目录被设置为像 `/dev/null` 的目录。

- 修复ecpg的SQLDA区域中 一些字段对齐问题(Zoltan Boszormenyi)

- 在ecpg `DEALLOCATE` 语句 中允许 `AT` 选项(Michael Meskes)

  支持这个的基础设施已有一段时间了， 但由于疏忽仍然有拒绝这种情况的错误检查。

- 当在ecpg中定义varchar结构时，不要使用变量名(Michael Meskes)

- 修复 `contrib/auto_explain` 的 JSON模式以产生有效的JSON(Andrew Dunstan)

  当它使用花括号时，该输出使用顶层方括号。

- 修复 `contrib/intarray` 的 `int[] &int[]` 操作符 中的错误(Guillaume Lelarge)

  如果最小整数的两个输入数组中常见的是1， 并且在其中之一数组中有较小的值，然后将 1 从结果中错误地省略。

- 修复 `contrib/pgcrypto` 的 `encrypt_iv()` 和 `decrypt_iv()` 中的错误检查(Marko Kreen)

  这些函数没有成功报告无效输入错误的某些类型， 并且反而为不正确输入返回随机垃圾值。

- 修复 `contrib/test_parser` 中一字节缓冲区超出范围(Paul Guyot)

  该代码将尝试读取比它应该的又一个字节，这将在困境情况下崩溃。 因为 `contrib/test_parser` 只是示例代码， 这本身不是一个安全问题，但不好的例子代码仍然是差的。

- 如果可用，那么 在ARM上为spinlocks使用 `__sync_lock_test_and_set()` (Martin Pitt)

  这个函数替换 `SWPB` 指令先前用法，它已经废弃并且在ARMv6和之后的不可用。 报告建议该旧代码在最新ARM上以显著方式使用， 但是不能简单地连锁并发访问，导致多进程操作中的离奇失败。

- 当编译接受它的gcc版本时， 使用 `-fexcess-precision=standard` 选项(Andrew Dunstan)

  这阻止各种各样的情节，其中gcc最新版本将产生创造性结果。

- 允许FreeBSD上线程Python的使用(Chris Rees)

  我们配置脚本先前认为这种组合不会允许；但是FreeBSD修复该问题， 因此删除错误检查。

- 允许MinGW建立使用标准命名OpenSSL库(Tomasz Ostrowski)

E.17. 发布9.1.2

> 发布日期: 2011-12-05

该发布包含来自9.1.1的各种修复。关于9.1主要版本新功能的信息， 参阅Section E.19。

E.17.1. 迁移到版本9.1.2

为运行9.1.X不需要转储/恢复。

然而，在 `information_schema.referential_constraints` 视图的定义中 发现了一个长期错误。如果你依赖该视图的正确结果，那么 你应该像下面第一个更新记录项解释的替换它的定义。

另外，如果你使用 `citext` 数据类型，并且你 通过运行pg_upgrade从以前的主要版本升级，你应该运行 `CREATE EXTENSION citext FROM unpackaged` 以避免 `citext` 操作中排序规则相关的错误。 如果你从包含 `citext` 数据类型实例的9.1之前数据库恢复转储，那么同样是必要的。如果在升级到9.1.2之前你已经运行 `CREATE EXTENSION` 命令， 你将不需要做手动更新目录， 正如第二个记录项解释的那样。

E.17.2. 变化

- 修复 `information_schema.referential_constraints` 视图中错误(Tom Lane)

  该视图对于匹配依赖主键的外键约束或者唯一性约束不够仔细。 这可能导致显示所有外键约束的错误，或者显示多次，或者 声明它取决于比确实存在的不同约束。

  因为该视图定义是通过initdb安装的， 只是升级不会修复该问题。如果你需要在现有的安装中修复这个问题， 你可以（作为一个超级用户）删除 `information_schema` 模式，然后通过 `_SHAREDIR_` /information*schema.sql* 重新创建它。 （如果你不确定 `*_SHAREDIR* `在哪里，运行` pg_config --sharedir`） 必须在被修复的每个数据库中重复。

- 使得 `contrib/citext` 的更新脚本 修复 `citext` 列和索引的排序规则(Tom Lane)

  现有的 `citext` 列和索引不能正确标记为 在pg_upgrade中来自 9.1之前服务器的collatable数据类型， 或者当包含 `citext` 类型的9.1之前备份被加载到9.1服务器时。 这导致这些列上的操作有错误而失败， 比如"不能决定为字符串比较使用哪个排序规则"。 这种变化可以通过 在 `CREATE EXTENSION citext FROM unpackaged` 中 升级 `citext` 模块到一个适当的9.1扩展的 相同脚本进行修复。

如果你有遇到这个问题的以前升级数据库， 而且你已经运行 `CREATE EXTENSION` ， 你可以手动运行（作为超级用户） 在 `_SHAREDIR_` /extension/citext--unpackaged--1.0.sql 结尾发现的 `UPDATE` 命令。 （如果你不确定 `_SHAREDIR_` 在哪，那么运行 `pg_config --sharedir` 。） 如果不确定再次这样做是没有害处的。

- 修复 `UPDATE` 或者 `DELETE` 加入到标量返回函数输出中的 可能崩溃(Tom Lane)

  如果目标行同时被更新，那么可能发生崩溃，因此 这个问题间歇性地出现。

- 修复GIN索引更新WAL记录的错误回放(Tom Lane)

  这可能导致在崩溃后或者热备服务器上暂时无法找到索引项， 然而，该问题可以通过索引的下一个 `VACUUM` 被修复。

- 修复 `CREATE TABLE dest AS SELECT * FROM src` 或者 `INSERT INTO dest SELECT * FROM src` 期间 TOAST相关数据损坏(Tom Lane)

  如果表通过 `ALTER TABLE ADD COLUMN` 被修改， 那么尝试逐字拷贝它的数据到另一个表在某些困境情况下可以产生崩溃结果。 该问题表现在8.4以及之后版本的精确形式中， 但是我们补丁早期版本以及有其他编码路径下可以触发相同错误。

- 修复热备启动中可能错误(Simon Riggs)

- 当初始快照不完整时，更快启动热备(Simon Riggs)

- 修复toast表访问陈旧syscache项中的竞争条件(Tom Lane)

  典型症状是短暂错误像"为pg_toast_2619中toast值NNNNN丢失块号0"， 其中引用的toast表总是从属于一个系统目录。

- 跟踪用于参数缺省表达式函数依赖(Tom Lane)

  以前，被引用的对象没有删除或者修改函数而被删除， 当使用该函数时，导致错误操作。 请注意，仅仅安装此更新将不能修复丢失依赖项； 这样，你之后需要 `CREATE OR REPLACE` 每个函数。 如果你有缺省依赖非内置对象的函数，这样做是值得推荐的。

- 修复nestloop连接中占位符变量的错误管理(Tom Lane)

  这个错误已知的导致"在子计划目标列中没有找到变量"规划器错误， 并且当涉及到外部连接时，可能导致错误查询输出。

- 修复涉及聚集表达式排序的window函数(Tom Lane)

  以前这些可能伴随"没有找到pathkey项排序"规划器错误而失败。

- 修复"MergeAppend子目录列不匹配MergeAppend"规划器错误(Tom Lane)

- 修复collatable和noncollatable输入索引匹配操作符(Tom Lane)

在9.1.0中，可索引操作符具有非collatable左边 输入类型和collatable右边输入类型不会被公认为匹配左边列的索引。 例子是 `hstore` `?` `text` 操作符。

- 允许有多个OUT参数的设置返回SQL函数的内联(Tom Lane)
- 不能信任连接删除的延缓唯一索引 (Tom Lane 和Marti Raudsepp)

  延迟唯一性约束可能不会持有内部事务， 因此假设它可以给定错误的查询结果。

- 使得 `DatumGetInetP()` 解压有1字节头的inet数据， 并且添加一个新宏，`DatumGetInetPP()` 确实没有(Heikki Linnakangas)

  这个变化不影响核心代码，但是 可能阻止希望 `DatumGetInetP()` 按惯例产生解压数据的附加代码中崩溃。

- 提高 `money` 类型的输入和输出的区域支持(Tom Lane)

  除了不支持所有标准的 `lc_monetary` 格式选项，输入和输出函数是一致的， 意味着有区域备份 `money` 值不能被重读。

- 不要让 `transform_null_equals` 影响 `CASE foo WHEN NULL ...` 结构 (Heikki Linnakangas)

  `transform_null_equals` 只会 影响直接由用户编写的 `foo = NULL` 表达式， 通过 `CASE` 这种形式内部产生的不平等检查。

- 改变外键触发器创建顺序更好地支持自我参照外键(Tom Lane)

  一个级联外键引用它自己的表，行更新 将触发 `ON UPDATE` 触发器和 作为一个事件的 `CHECK` 。 `ON UPDATE` 触发器必须首先执行，否则 `CHECK` 将检查 该行的非最终状态并且可能抛出一个不合适错误。 然而，这些触发器的触发顺序是由自己名字决定的， 其中通常按照创建顺序排序，因为 触发器按照惯例"RI_ConstraintTrigger_NNNN" 有自动生成的名字。 一个适当的修复将需要修改该惯例， 我们会在9.2中执行，但在现有的版本中改变它似乎有风险。 所以这个补丁只改变 触发器的创建顺序。用户遇到此类型的错误要删除并重新创建外键约束使得它的 触发器进入正确的顺序。

- 修复 `DROP OPERATOR FAMILY` 中 `IF EXISTS` 正常运行(Robert Haas)
- 不允许来自自己脚本中扩展的删除(Tom Lane)

  这阻止了在扩展依赖的错误管理下的古怪操作。

- 不要标记自动生成类型为扩展成员(Robert Haas)

  关系行类型以及自动生成的数组类型不需要 `pg_depend` 中自己的扩展成员项， 并且创建这样的项使得扩展更新复杂化。

- 在 `CREATE EXTENSION` 中处理无效的早已存在的 `search_path` 设置(Tom Lane)
- 当跟踪缓冲区分配率时，避免浮点下溢(Greg Matthews)

当对自身无害时，在某些平台上这可能导致讨厌的内核日志信息。

- 防止可串行化模式下运行的自动清理事务(Tom Lane)

  以前自动清理使用集群端缺省事务隔离级别，但是 没有必要使用高于READ COMMITTED的任何东西， 并且使用SERIALIZABLE可以导致其他进程的不必要延迟。

- 确保walsender进程迅速反应给SIGTERM(Magnus Hagander)
- 从基础备份中排除 `postmaster.opts` (Magnus Hagander)
- 当在Windows下启动子进程时，保留配置文件名字和行号值(Tom Lane)

  以前，这些在 `pg_settings` 视图中不能被正确显示。

- 修复ecpg的SQLDA区域中不正确字段对齐(Zoltan Boszormenyi)
- 保留psql的命令历史中该命令中的空白行(Robert Haas)

  如果从字符串中删除空行，前者操作可能产生问题，比如。

- 避免pg_dump中特定平台无限循环(Steve Singer)
- 修复pg_dump中纯文本输出格式的压缩 (Adrian Klaver和Tom Lane)

  pg_dump从历史角度理解没有 `-F` 切换的 `-Z` ， 这意味着它应该发出纯文本输出的gzip压缩版本。恢复该行为。

- 修复pg_dump以备份自动生成类型之间用户定义的映射， 比如表rowtype(Tom Lane)
- 修复pg_dump中外服务器名字的丢失引用(Tom Lane)
- pg_upgrade各种修复(Bruce Momjian)

  正确处理排斥约束，避免Windows上错误， 不要抱怨8.4数据库中不匹配toast表名。

- 在PL/pgSQL中，允许外表定义行类型(Alexander Soudakov)
- 解决了PL/Perl函数结果转换(Alex Hunsaker和Tom Lane)

  恢复PL/Perl函数返回 `void` 的9.1之前操作忽略了最后Perl语句的结果值； 如果该语句返回一个引用，那么9.1.0可能抛出错误。 另外，确保它返回复合类型字符串值， 只要该字符串符合类型的输入格式。 此外，当函数的声明结果类型分别不是数组或者复合类型时， (先前9.1版本而不能返回字符串像这种情况中的 `ARRAY(0x221a9a0)` 或者 `HASH(0x221aa90)`) 尝试返回Perl数组或者哈希而抛出错误。

- 确保PL/Perl字符串总是正确的UTF8编码(Amit Khandekar和Alex Hunsaker)
- 使用xsubpp首选版本以编译PL/Perl， 不一定操作系统的主拷贝(David Wheeler和 Alex Hunsaker)

- 在PL/Python异常中正确扩散SQLSTATE(Mika Eloranta和Jan Urbanski)
- 为Python主要版本不同于一个建立的不要安装PL/Python扩展文件(Peter Eisentraut)
- 如果他们提供psql，那么 改变所有 `contrib` 扩展脚本文件以报告 有用错误消息(Andrew Dunstan和Tom Lane)

  这有助于教会人们关于使用 `CREATE EXTENSION` 的新方法 加载这些文件。在多数情况下，使用的脚本可能直接失败，但是 伴随着难以解释信息。

- 修复 `contrib/dict_int` 和 `contrib/dict_xsyn` 中错误编码(Tom Lane)

  一些函数错误地假设通过 `palloc()` 返回的内存保证为零。

- 从正则表达式测试机制中删除 `contrib/sepgsql` 测试(Tom Lane)

  因为这些测试需要root权限，它们不切实际地自动运行。 相反切换到手动方式，并且提供测试脚本。

- 修复 `contrib/unaccent` 的配置文件解析中的各种错误(Tom Lane)
- 接受 `pgstatindex()` 中的及时查询取消中断(Robert Haas)
- 修复Mac OS X启动脚本中日志文件名的错误引用(Sidar Lopez)
- 恢复 `WAL_DEBUG` 的意外激活(Robert Haas)

  幸运的是，作为调试工具，这是相当便宜的；但它并不打算缺省启用，所以恢复。

- 确保VPATH编译正确安装所有服务器头文件(Peter Eisentraut)
- 缩短详细错误消息中报告的文件名(Peter Eisentraut)

  规则编译一直被包含错误消息调用的C文件名报告， 但是VPATH编译之前报告绝对路径名。

- 修复中美洲Windows时区名解释(Tom Lane)

  映射"中美洲标准时间"为 `CST6` ，而不是 `CST6CDT` ， 因为在中美洲任何地方通常观察不到DST。

- 为了Brazil, Cuba, Fiji, Palestine, Russia和Samoa中DST变化规律 更新时区数据文件到tzdata发布2011n；以及历史修正Alaska和British East Africa。

E.18. 发布9.1.1

> 发布日期: 2011-09-26

该发布包含了9.1.0少量修复。关于9.1主要发布的新功能的信息， 参阅Section E.19。

E.18.1. 迁移到版本9.1.1

运行9.1.X不需要备份/恢复。

E.18.2. 变化

- 使得 `pg_options_to_table` 为没有值的选项返回空(Tom Lane)

  以前这种情况可以导致服务器崩溃。

- 修复GiST索引扫描末尾的内存泄露(Tom Lane)

  执行许多单独GiST索引扫描的命令， 比如包含很多行的表上新的基于排斥约束的GiST验证， 由于这种泄露可能短暂地需要大量内存。

- 修复 `CREATE TEMPORARY TABLE` 中 显示引用到 `pg_temp` 模式(Robert Haas)

  这过去是被允许的，但是在9.1.0中失败。

E.19. 发布9.1

> 发布日期: 2011-09-12

E.19.1. 概述

该发布显示了PostgreSQL超越传统关系数据库功能设置为新， 突破性功能对PostgreSQL是唯一的。 发布9.0中介绍的流复制功能通过添加同步复制选项，流备份以及 监测改进而被显著增强。主要功能包含：

- 允许同步复制
- 添加支持外表
- 添加每列collation支持
- 添加扩展 这简化附加包到PostgreSQL。
- 添加真的可串行化隔离级别
- 在 `CREATE TABLE` 中 使用 `UNLOGGED` 选项支持未记录表
- 允许 `WITH` 子句中 数据修改命令(`INSERT` / `UPDATE` / `DELETE`)
- 添加最近邻（算子排序）搜索到 GiST索引
- 添加 `SECURITY LABEL` 命令并且支持SELinux权限控制
- 更新PL/Python服务器端语言

在下面部分将详细解释上面项。

E.19.2. 迁移到版本9.1

使用pg_dump或者pg_upgrade 备份/恢复， 对于那些希望从任何以前发布中迁移数据是必须的。

版本9.1包含了可能影响与以前版本的兼容性的一些变化。 观察下面的不兼容性：

E.19.2.1. 字符串

- 改变 `standard_conforming_strings` 的缺省值(Robert Haas)

缺省情况下，反斜杠是字符串中普通字符， 而不是转义字符。这一变化删除与SQL标准长期存在的不兼容。 `escape_string_warning` 有多年关于这个用法的警告。 `E''` 字符串是嵌入反斜杠转义到字符串中的正确做法， 并且这种变化不受影响。

Warning

这个变化可以打断不希望的应用，并且按照旧的规则执行字符串逃逸。 该结果可能与介绍的SQL注入安全中断一样严峻。 确保测试应用遭受不可信任输入， 确保它们正确处理单引号和文本字符串中反斜线符号。

E.19.2.2. 转换

- 不允许函数样式和属性样式数据类型转换为复合类型(Tom Lane)

  比如，不允许 `_composite_value_`.text和 `text(``_composite_value_`)。 该语法的无意识使用经常导致错误报告； 尽管它不是错误，似乎追溯到拒绝这样的表达式更好。 当实际上打算整个复合值计算时， `CAST` 和 `::` 语法仍然可以使用。

- 加强基于数组的域转换检查(Tom Lane)

  当域是基于数组类型时，它被允许"查看"域类型访问数组元素， 包含下标域值抓取或者分配一个元素。 分配这样一个域值元素，比如通过 `UPDATE ... SET domaincol[5] = ...`， 将导致检查域类型的约束，然而检查之前被忽略。

E.19.2.3. 数组

- 改变 `string_to_array()` 为零长字符串返回空数组(Pavel Stehule)

  以前返回空值。

- 改变 `string_to_array()`， 所以 `NULL` 分隔符分离字符串到字符(Pavel Stehule)

  先前这返回空值。

E.19.2.4. 对象修改

- 修复触发器之前/之后的不当检查(Tom Lane)

  触发器可以在三种情况下被触发： `BEFORE`， `AFTER` 或者 `INSTEAD OF` 一些操作。 触发器函数发起者应该理智验证三种情况下它们的逻辑操作。

- 需要超级用户或者 `CREATEROLE` 权限目的是设置评论角色(Tom Lane)

E.19.2.5. 服务器设置

- 改变 `pg_last_xlog_receive_location()` 因此它不会向后移动(Fujii Masao)

  先前，当重新启动流复制的时候，`pg_last_xlog_receive_location()` 的值 可以向后移动。

- 复制连接日志接受 `log_connections` (Magnus Hagander)

  先前，复制连接总是被记录。

E.19.2.6. PL/pgSQL服务器端语言

- 改变不带参数的PL/pgSQL的 `RAISE` 命令 为通过附属异常块可捕获(Piyush Newe)

  以前代码块中 `RAISE` 总是限于附加异常块中，因此 在同一范围内是无法捕获的。

- 调整PL/pgSQL的错误线编号代码与其他PL相一致(Pavel Stehule)

  先前，PL/pgSQL在函数体开始部分可能忽略（不计算）空行。 由于这是不符合所有其他语言的，特殊情况下被删除。

- 使得PL/pgSQL抱怨IN和OUT参数名冲突(Tom Lane)

  以前，检测不到冲突，该名字可能就默默地指向OUT参数。

- PL/pgSQL变量类型修饰符对于SQL分析器是可见的(Tom Lane)

  附属于PL/pgSQL变量的类型修饰符（比如varchar长度限制） 在分配之间被执行，但是出于其他目的被忽略。 这些变量操作更像声明了同一描述符的表列。 这不希望在多数情况下产生任何可见差异， 但是它可能产生通过PL/pgSQL函数发布的一些SQL命令的微妙变化。

E.19.2.7. Contrib

- 所有contrib模块伴随 `CREATE EXTENSION` 被安装而不是通过手动调用它们的SQL脚本(Dimitri Fontaine, Tom Lane)

  为了更新包含contrib模块9.0版本的现有数据库， 使用 `CREATE EXTENSION ... FROM unpackaged` 封装现有contrib模块的对象到一个扩展中。 当从先前9.0版本中更新时，使用旧的卸载脚本删除contrib模块的对象， 然后使用 `CREATE EXTENSION` 。

E.19.2.8. 其他的不兼容

- 使得 `pg_stat_reset()` 重置所有数据库级别统计(Tomas Vondra)

  一些 `pg_stat_database` 计数器没有被重置。

- 修复一些 `information_schema.triggers` 列名匹配新的SQL标准名(Dean Rasheed)
- 把ECPG游标名看作不区分大小写(Zoltan Boszormenyi)

E.19.3. 变化

下面你将发现在PostgreSQL 9.1 和以前主要发布之间的变化的详细列表。

E.19.3.1. 服务器

E.19.3.1.1. 性能

- 支持 在 `CREATE TABLE` 中 使用 `UNLOGGED` 选项的未记录表(Robert Haas)

  这个表提供了比常规表更好的更新性能，但是不安全：它们内容在 服务器崩溃的情况下被自动清除。 也不会被传送到备份子服务器。

- 允许 `FULL OUTER JOIN` 作为哈希连接被实现， 并且允许 `LEFT OUTER JOIN` 或者 `RIGHT OUTER JOIN` 的一侧被哈希(Tom Lane)

  以前 `FULL OUTER JOIN` 只能作为合并连接被实现， 并且 `LEFT OUTER JOIN` 和 `RIGHT OUTER JOIN` 可以哈希连接空侧。 这些变化提供了额外查询优化可能性。

- 合并复制fsync请求(Robert Haas, Greg Smith)

  这大大提高了大量写入重载下性能。

- 提高 `commit_siblings` 的性能(Greg Smith)

  这允许较少开销的 `commit_siblings` 的使用。

- 减少大的ispell词典存储需求(Pavel Stehule, Tom Lane)
- 避免"盲写"后将数据文件打开(Alvaro Herrera)

  这修复它们被删除后后端保持文件打开时间，防止内核回收磁盘空间。

E.19.3.1.2. 优化器

- 允许继承表扫描返回有意义的排序结果 (Greg Stark, Hans-Jurgen Schonig, Robert Haas, Tom Lane)

  这允许使用 `ORDER BY` ， `LIMIT` 或者带有继承表的 `MIN` / `MAX` 的查询的更好优化。

- 提高GIN索引扫描成本估计(Teodor Sigaev)
- 提高聚集和window函数成本估计(Tom Lane)

E.19.3.1.3. 认证

- 在 `pg_hba.conf` 中支持host名字和host后缀(比如 `.example.com`)

  先前只支持host IP 地址和CIDR值。

- 在 `pg_hba.conf` 的host列 支持关键字 `all` (Peter Eisentraut)

  以前为了这个人们使用 `0.0.0.0/0` 或者 `::/0` 。

- 在不支持Unix套接字连接的平台上拒绝 `pg_hba.conf` 中的 `local` 行(Magnus Hagander)

  先前，默默忽略这样的行，这可能是令人惊讶的。 这使得该操作更像其他不支持的情况。

- 允许GSSAPI 通过SSPI用于 验证到服务器(Christian Ullrich)

  特别是允许基于Unix GSSAPI客户端执行 伴随Windows服务器的SSPI认证。

- 在区域套接字上的 `ident` 认证被称为 `peer` (Magnus Hagander)

  该旧的术语仍然接受向后兼容性，但是 因为这两种方法在根本上是不同的， 为它们采用不同的名字似乎更好。

- 重写peer认证 以避免证书控制信息的使用(Tom Lane)

  这个变化使得同等验证码更加简单而且可执行性更好。然而， 它需要平台提供 `getpeereid` 函数或者等效套接字操作。 按目前所知，之前同等认证执行唯一平台，并且现在不是之前5.0 NetBSD。

E.19.3.1.4. 监控

- 添加重启点和检查点记录详情，通过 `log_checkpoints` 控制 (Fujii Masao, Greg Smith)

  新的详情包含WAL文件以及同步活动。

- 添加 `log_file_mode` 其控制通过日志收集器创建的日志文件权限(Martin Pihlak)

- 减少syslog记录缺省最大行长度 到加上前缀900字节(Noah Misch)

  这避免了syslog实现上具有1KB长度限制而不是常见的2KB的长日志行的 截断。

E.19.3.1.5. 统计视图

- 添加 `client_hostname` 列到 `pg_stat_activity` (Peter Eisentraut)

  先前只报道客户端地址。

- 添加 `pg_stat_xact_*` 统计函数和视图(Joel Jacobson)

  这些更像数据库端统计计算视图， 但是反映了当前事务重要性。

- 添加数据库级别最后重置时间以及后端写统计视图(Tomas Vondra)
- 在 `pg_stat_*_tables` 视图中添加列显示清理数和分析操作(Magnus Hagander)
- 添加 `buffers_backend_fsync` 列到 `pg_stat_bgwriter` (Greg Smith)

  这个新列计算后端同步缓冲区的次数。

E.19.3.1.6. 服务器设置

- 提供 `wal_buffers` 的自动调整(Greg Smith)

  缺省情况下，`wal_buffers` 的值基于 `shared_buffers` 的值 被自动选择。

- 为 `deadlock_timeout`， `log_min_duration_statement` 和 `log_autovacuum_min_duration` 增加最大值(Peter Eisentraut)

  这些参数的最大值先前只有35分钟。 现在允许更大值。

E.19.3.2. 备份与恢复

E.19.3.2.1. 流复制和连续归档

- 允许同步复制 (Simon Riggs, Fujii Masao)

  这允许等待备用的主服务器在承认提交前写事务信息到磁盘。 每次备用起着同步备份的作用。正如通过 `synchronous_standby_names` 设置控制。 在每个事务基础上使用 `synchronous_commit` 设置可以启用或者禁用同步复制。

- 使用流复制网络连接添加发送文件系统备份协议支持到备用服务器 (Magnus Hagander, Heikki Linnakangas)

  当设置一个备用服务器时，这避免了手动传送文件系统备份的需求。

- 添加 `replication_timeout` 设置(Fujii Masao, Heikki Linnakangas)

  对于超过 `replication_timeout` 间隔空闲的复制连接将自动被终止。 先前，失败连接通常不能被检测直到TCP超时消逝， 在很多情况下这是不方便的。

- 添加命令行工具 pg_basebackup 创建一个新的备用服务器或者数据库备份(Magnus Hagander)
- 为角色添加复制许可(Magnus Hagander)

  这是用于流复制的只读权限。它允许非超级用户 用于复制连接。 先前只有超级用户可以启动复制连接； 缺省情况下超级用户具有此权限。

E.19.3.2.2. 备份监控

- 增加系统视图 `pg_stat_replication`， 这显示了WAL发送者进程活动 (Itagaki Takahiro, Simon Riggs)

  报告所有连接备用服务器的状态。

- 添加监控函数 `pg_last_xact_replay_timestamp()` (Fujii Masao)

  这返回了主库产生最近提交或者应用在备库上的终止记录的时间。

E.19.3.2.3. 热备份

- 添加配置参数 `hot_standby_feedback` 启动备库以推迟主库旧行版本清理(Simon Riggs)

  这有助于避免备库上取消长时间运行查询。

- 添加 `pg_stat_database_conflicts` 系统视图以显示取消的查询和原因(Magnus Hagander)

  因为删除的表空间， 锁超时， 旧快照， 保留区以及死锁， 可以发生取消。

- 添加 `conflicts` 计算到 `pg_stat_database` (Magnus Hagander)

  发生在数据库中的冲突数。

- 为 `max_standby_archive_delay` 和 `max_standby_streaming_delay` 增加最大值。

  每个参数的最大值以前只有35分钟。现在允许更大的值。

- 添加 `ERRCODE_T_R_DATABASE_DROPPED` 错误代码以报告恢复冲突， 由于已删除数据库 (Tatsuo Ishii)

  这对于连接池软件是有用的。

E.19.3.2.4. 恢复控制

- 添加函数以控制流复制重播(Simon Riggs)

  新函数是 `pg_xlog_replay_pause()` , `pg_xlog_replay_resume()` , 并且状态函数是 `pg_is_xlog_replay_paused()`

- 添加 `recovery.conf` 设置 `pause_at_recovery_target` 在目标位置暂停恢复(Simon Riggs)

  这允许需要恢复服务器检查恢复点是否是所需的。

- 使用 `pg_create_restore_point()` 添加创建命名恢复点的能力(Jaime Casanova)

  这些命名恢复点被声明为使用新的 `recovery.conf` 设置 `recovery_target_name` 的恢复目标。

- 允许备份恢复自动切换到新的时间线(Heikki Linnakangas)

  目前备用服务器为周期性新的时间线扫描归档目录。

- 添加 `restart_after_crash` 设置，在后台崩溃后禁用自动服务器重启(Robert Haas)

  这允许外部集群管理软件控制数据库服务器是否重新启动。

- 允许 `recovery.conf` 使用和 `postgresql.conf` 相同的引用操作(Dimitri Fontaine)

  以前所有值必须被引用。

E.19.3.3. 查询

- 添加真的 可串行化隔离级别 (Kevin Grittner, Dan Ports)

  先前，要求串行化隔离保证了单独MVCC快照可以用于整个事务， 这允许某些记录异常。 旧的快照隔离级别通过请求 `REPEATABLE READ` 隔离级别可用。

- 允许 `WITH` 子句中 数据修改命令(`INSERT` / `UPDATE` / `DELETE`) (Marko Tiikkaja, Hitoshi Harada)

  这些命令可以使用 `RETURNING` 传递数据到所包含查询。

- 允许 `WITH` 子句被附属于 `INSERT`， `UPDATE`， `DELETE` 语句(Marko Tiikkaja, Hitoshi Harada)

- 当在 `GROUP BY` 子句中指定主键时， 允许查询目标列中的 非 `GROUP BY` 列(Peter Eisentraut)

  SQL标准允许该操作，由于该主键，结果是明确的。

- 允许 `UNION` / `INTERSECT` / `EXCEPT` 子句中关键字 `DISTINCT` 的使用(Tom Lane)

  `DISTINCT` 是缺省操作，因此该关键字的使用是多余的， 但是SQL标准允许它。

- 修复普通规则查询以使用和 `EXPLAIN ANALYZE` 相同快照操作(Marko Tiikkaja)

  先前 `EXPLAIN ANALYZE` 使用涉及规则的查询略微不同的快照时间。 `EXPLAIN ANALYZE` 操作被认为更合乎逻辑。

E.19.3.3.1. 字符串

- 添加每列 collation支持 (Peter Eisentraut, Tom Lane)

  以前在数据库创建中选择排序规则（文本字符串排序）。 排序规则可以设置每列，域，索引，或者表达式， 通过SQL标准 `COLLATE` 子句。

E.19.3.4. 对象操作

- 添加extensions， 这简化了附加PostgreSQL的包装(Dimitri Fontaine, Tom Lane)

通过新的 `CREATE` / `ALTER` / `DROP EXTENSION` 命令 控制扩展。 这取代了分组对象的点对点方法被添加到PostgreSQL安装中。

- 添加支持外表 (Shigeru Hanada, Robert Haas, Jan Urbanski, Heikki Linnakangas)

  这允许存储在数据库外的数据像本地PostgreSQL存储数据一样使用。 然而，外表目前只读。

- 允许新值通过 `ALTER TYPE` 被 添加到现有枚举类型中(Andrew Dunstan)
- 添加 `ALTER TYPE ... ADD/DROP/ALTER/RENAME ATTRIBUTE` (Peter Eisentraut)

  这允许复合类型的修改。

E.19.3.4.1. `ALTER` 对象

- 在分类表上添加 `RESTRICT` / `CASCADE` 到 `ALTER TYPE` 操作(Peter Eisentraut)

  这控制 `ADD` / `DROP` / `ALTER` / `RENAME ATTRIBUTE` 级联操作。

- 支持 `ALTER TABLE` `_name_` {OF | NOT OF} `_type_` (Noah Misch)

  这个语法允许单独表成为分类表，或者分类表成为单独的。

- 在 `ALTER ... SET SCHEMA` 命令中添加支持多个对象类型(Dimitri Fontaine)

  该命令现在支持转换，算子，算子类， 算子族，文本搜索配置，文本搜索词典，文本 搜索分析器，以及文本搜索模板。

E.19.3.4.2. `CREATE/ALTER TABLE`

- 添加 `ALTER TABLE ... ADD UNIQUE/PRIMARY KEY USING INDEX` (Gurjeet Singh)

  这允许使用现有唯一索引包含 同时创建的唯一索引定义主键或者唯一约束。

- 允许 `ALTER TABLE` 添加没有验证的外键(Simon Riggs)

  新选项称为 `NOT VALID` 。 约束的状态可以随后被修改为 `VALIDATED` 并且 执行验证检查。 同时这些允许你添加对读写操作影响最小的外键。

- 允许 `ALTER TABLE... SET DATA TYPE` 避免 在合适情况中的表重写(Noah Misch, Robert Haas)

  比如，转换 `varchar` 列使得 `text` 不再需要表重写。 然而，增加 `varchar` 列上长度限制仍然需要表重写。

- 添加 `CREATE TABLE IF NOT EXISTS` 语法(Robert Haas)

  如果该表已经存在，那么允许没有导致错误的表创建。

- 当两个后端尝试添加继承孩子到同一时间的同一表时， 修复可能"元组同时更新"错误(Robert Haas)

  `ALTER TABLE` 需要采取更强锁定父表，这样该会话不能试图同时更新它。

E.19.3.4.3. 对象权限

- 添加 `SECURITY LABEL` 命令(KaiGai Kohei)

  这允许安全标签分配给对象。

E.19.3.5. 实用操作

- 添加事务级别咨询锁(Marko Tiikkaja)

  这与现有会话级别咨询锁类似， 但是在事务结束时自动释放这种锁。

- 使得 `TRUNCATE ... RESTART IDENTITY` 相互作用的重启序列(Steve Singer)

  如果在提交事务活动和提交完成之间发生后端崩溃， 那么该计数器可能已经不同步。

E.19.3.5.1. `COPY`

- 添加 `ENCODING` 选项到 `COPY TO/FROM` (Hitoshi Harada, Itagaki Takahiro)

  这允许分别从客户端编码指定 `COPY` 文件编码。

- 添加双向 `COPY` 协议支持(Fujii Masao)

  当前通过流复制被使用。

E.19.3.5.2. `EXPLAIN`

- 使得 `EXPLAIN VERBOSE` 在 `FunctionScan` 节点中 显示函数调用表达式(Tom Lane)

E.19.3.5.3. `VACUUM`

- 添加额外详情到 `VACUUM FULL VERBOSE` 和 `CLUSTER VERBOSE` 的输出中(Itagaki Takahiro)

  新信息包含活的和死的元组数， 以及是否 `CLUSTER` 使用索引重新创建。

- 如果它不能获得表锁， 那么阻止autovacuum等待(Robert Haas)

  它将尝试随后清理该表。

E.19.3.5.4. `CLUSTER`

- 当它似乎比较廉价时，允许 `CLUSTER` 排序表而不是 扫描索引(Leonardo Francalanci)

E.19.3.5.5. 索引

- 添加最近邻（算子排序）搜索到 GiST索引 (Teodor Sigaev, Tom Lane)

  这允许GiST索引迅速返回带有 `LIMIT` 的查询中的 `_N_` 最近值。比如

  ```
  SELECT * FROM places ORDER BY location &lt;-&gt; point '(101,456)' LIMIT 10;
  ```

  找到离给定目标点最近的十个位置。

- 允许GIN索引索引无效和空值(Tom Lane)

  这允许全GIN索引扫描， 并且修复GIN扫描可能失败的情况。

- 允许GIN索引 更好识别重复搜索项(Tom Lane)

  这降低了索引扫描成本， 尤其是在避免不必要全扫描索引的情况下。

- 修复GiST索引充分 碰撞安全(Heikki Linnakangas)

  先前存在需要 `REINDEX` 的罕见情况（你可能被告知）。

E.19.3.6. 数据类型

- 允许 `numeric` 在一般情况下使用更紧凑，两字节标题。

  以前所有 `numeric` 值有四字节标题， 这种变化节省磁盘存储。

- 添加通过 `money` 划分 `money` 的支持(Andy Balholm)

- 允许类型 `void` 上的二进制I/O(Radoslaw Smogura)

- 提高几何学操作符的斜边计算(Paul Matthews)

  这避免了不必要的溢出，也可能更精准。

- 支持哈希数组值(Tom Lane)

  这提供了额外查询优化可能性。

- 不要将复合类型看作可分类的除非所有列类型是可分类的(Tom Lane)

  这避免了可能的运行时"不能识别比较函数"错误， 如果可能实现没有排序的查询。 同时， `ANALYZE` 不会尝试为该复合类型列使用不恰当的统计收集方法。

E.19.3.6.1. Casting

- 添加支持 `money` 和 `numeric` 之间的转换(Andy Balholm)

- 添加从 `int4` 和 `int8` 到 `money` 的转换(Joey Adams)

- 如果它是一个分类表， 那么允许转换表的行类型到表的父类型(Peter Eisentraut)

  这与允许转换行类型到父表行类型的现有功能类似。

E.19.3.6.2. XML

- 添加XML函数 `XMLEXISTS` 和 `xpath_exists()` 函数(Mike Fowler)

  这些用于XPath匹配。

- 添加XML函数 `xml_is_well_formed()`, `xml_is_well_formed_document()`, `xml_is_well_formed_content()` (Mike Fowler)

  这些检查输入是否是恰当形成的XML。 它们提供了以前在弃用的 `contrib/xml2` 模块中可用的功能。

E.19.3.7. 函数

- 添加函数 `format(text, ...)`， 这类似于C的 `printf()` (Pavel Stehule, Robert Haas)

  它目前支持字符串，SQL文本以及SQL标识符格式。

- 添加字符串函数 `concat()`, `concat_ws()`, `left()`, `right()`, 以及 `reverse()` (Pavel Stehule)

  这改善了与其他数据库产品的兼容性。

- 添加函数 `pg_read_binary_file()` 以读取二进制文件(Dimitri Fontaine, Itagaki Takahiro)
- 添加函数 `pg_read_file()` 的一个参数版本以读取整个文件(Dimitri Fontaine, Itagaki Takahiro)
- 为空值处理控制添加 `array_to_string()` 和 `string_to_array()` 的三个参数形式(Pavel Stehule)

E.19.3.7.1. 对象信息函数

- 添加 `pg_describe_object()` 函数(Alvaro Herrera)

  该函数用于获得人类可读字符串描述对象，基于 `pg_class` OID， 对象OID以及子对象OID。它可以有助于解释 `pg_depend` 的内容。

- 更新内置操作符注释和潜在函数(Tom Lane)

  打算通过相关算子使用的函数同样被注释。

- 添加函数 `quote_all_identifiers` 强迫 `EXPLAIN` 中以及系统目录函数像 `pg_get_viewdef()` 中所有标识符引用。

尝试输出模式到工具和伴随不同引用规则更加容易的其他数据库。

- 添加列到 `information_schema.sequences` 系统表(Peter Eisentraut)

  先前，尽管视图存在，未实现关于序列参数的列。

- 在 `has_table_privilege()` 中以及相关函数中允许 `public` 作为伪角色名字(Alvaro Herrera)

  这允许检查public权限。

E.19.3.7.2. 函数和触发器创建

- 支持视图上 `INSTEAD OF` 触发器(Dean Rasheed)

  该功能可以充分实现可更新视图。

E.19.3.8. 服务器端语言

E.19.3.8.1. PL/pgSQL服务器端语言

- 添加 `FOREACH IN ARRAY` 到PL/pgSQL(Pavel Stehule)

  这比遍历数组值元素的先前方法更有效并且可读性更高。

- 允许可以从同一地方捕获 `RAISE ERROR` 的同一位置 捕获不带参数的 `RAISE` (Piyush Newe)

  前面编码从包含活跃异常处理程序块中抛出错误。 新操作与其他DBMS产品更加一致。

E.19.3.8.2. PL/Perl服务器端语言

- 允许通用记录参数到PL/Perl函数(Andrew Dunstan)

  可以声明PL/Perl函数接受类型 `record` 。 该操作与任何命名复合类型是一样的。

- 转换PL/Perl数组参数到Perl数组(Alexey Klyukin,Alex Hunsaker)

  字符串表示仍然可用。

- 转换PL/Perl复合类型参数到Perl哈希(Alexey Klyukin, Alex Hunsaker)

  字符串表示仍然可用。

E.19.3.8.3. PL/Python服务器端语言

- 添加表函数支持PL/Python(Jan Urbanski)

  PL/Python现在可以返回多个 `OUT` 参数和记录集。

- 添加验证器给PL/Python (Jan Urbanski)

  这允许PL/Python函数在函数创建时间是语法检查的。

- 允许PL/Python中SQL查询异常(Jan Urbanski)

  这允许从PL/Python异常块中访问SQL生成异常错误代码。

- 添加明确子事务到PL/Python (Jan Urbanski)
- 为引用字符串添加PL/Python函数(Jan Urbanski)

  这些函数是 `plpy.quote_ident` ， `plpy.quote_literal` ， 和 `plpy.quote_nullable` 。

- 添加跟踪信息到PL/Python错误(Jan Urbanski)
- 从带有 `PLy_elog` 的迭代器中报告PL/Python错误(Jan Urbanski)
- 修复Python 3异常处理(Jan Urbanski)

  异常类以前在Python 3下 `plpy` 中不可用。

E.19.3.9. 客户端应用

- 标记 createlang 和droplang 为废弃的现在它们只是调用扩展命令(Tom Lane)

E.19.3.9.1. psql

- 增加psql命令 `\conninfo` 以显示当前连接信息(David Christensen)
- 添加psql命令 `\sf` 以显示函数的定义(Pavel Stehule)
- 添加psql命令 `\dL` 罗列语言(Fernando Ike)
- 添加 `S` ("system")选项到psql的 `\dn` (罗列模式)命令(Tom Lane)

  没有 `S` 的 `\dn` 现在抑制系统模式。

- 允许 psql的 `\e` 和 `\ef` 命令 接受用于定位编辑器中游标的行号(Pavel Stehule)

  按照 `PSQL_EDITOR_LINENUMBER_ARG` 环境变量被传递给编辑器。

- psql设置来自缺省操作系统区域的客户端编码(Heikki Linnakangas)

  如果没有设置 `PGCLIENTENCODING` 环境变量， 那么只会发生。

- 尝试 `\d` 区分唯一索引和唯一约束(Josh Kupershmidt)
- 当讨论9.0或者之后服务器时， 尝试 `\dt+` 报告 `pg_table_size` 而不是 `pg_relation_size` (Bernd Helmle)

这是表大小更加有效的度量， 但是注意这与同一显示中先前报告的是不一样的。

- 额外的tab实现支持 (Itagaki Takahiro, Pavel Stehule, Andrey Popp, Christoph Berg, David Fetter, Josh Kupershmidt)

E.19.3.9.2. pg_dump

- 添加pg_dump 和pg_dumpall 选项 `--quote-all-identifiers` 强制引用所有标识符(Robert Haas)

- 添加 `directory` 格式给pg_dump (Joachim Wieland, Heikki Linnakangas)

  内部类似于 `tar` pg_dump格式。

E.19.3.9.3. pg_ctl

- 修复pg_ctl， 所以它不再错误地报告服务器没有运行(Bruce Momjian)

  如果服务器正在运行但是pg_ctl无法验证， 先前这可能发生。

- 提高pg_ctl启动的"wait" (`-w`)选项(Bruce Momjian, Tom Lane)

  等待模式现在更加鲁棒， 它通过非缺省postmaster port号， 非缺省 Unix域套接字位置， 权限问题， 或者旧的postmaster锁文件不会被混乱。

- 添加 `promote` 选项到pg_ctl用来切换 备用服务器到主库(Fujii Masao)

E.19.3.10. 开发工具

E.19.3.10.1. libpq

- 添加libpq连接选项 `client_encoding` 这就像 `PGCLIENTENCODING` 环境变量(Heikki Linnakangas)

  值 `auto` 设置基于操作系统区域的客户端编码。

- 添加 `PQlibVersion()` 函数， 它返回libpq库版本(Magnus Hagander)

  libpq已有 `PQserverVersion()` 返回服务器版本。

- 当通过伴随新的 `requirepeer` 连接选项的Unix域套接字连接时， 允许libpq使用客户端检查服务器进程用户名。(Peter Eisentraut)

  当通过Unix域套接字连接时， PostgreSQL允许服务器检查 客户端用户名。

- 添加 `PQping()` 和 `PQpingParams()` 到libpq (Bruce Momjian, Tom Lane)

  这些函数允许不尝试打开新会话的服务器状态检测。

E.19.3.10.2. ECPG

- 允许ECPG接受 `WHERE CURRENT OF` 子句 中动态游标名称(Zoltan Boszormenyi)
- 尝试ecpglib写带有15位精度而不是以前的14位的 `double` 值(Akira Kurosawa)

E.19.3.11. 编译选项

- 使用可以接受它的HP-UX C编译器的 `+Olibmerrno` 编译标记(Ibrar Ahmed)

  这避免了最新HP平台上math库调用的可能的不当操作。

E.19.3.11.1. Makefiles

- 改善并行make支持(Peter Eisentraut)

  允许更快编译。同时， `make -k` 更加一致运行。

- 需要GNU make 3.80或者更新版本(Peter Eisentraut)

  由于并行make改进这是必须的。

- 添加 `make maintainer-check` 目标(Peter Eisentraut)

  这个目标执行各种不适用于编译或者回归测试的源代码检查。 目前，duplicate_oids，SGML语法和tab检查，NLS语法检查。

- 支持 `contrib` 中的 `make check` (Peter Eisentraut)

  以前只有 `make installcheck` 运行，但是现在 支持临时安装中测试。 顶级 `make check-world` 包含这种方式测试 `contrib` 。

E.19.3.11.2. Windows

- 在Windows上，允许pg_ctl 注册服务为自动启动或者是按需启动(Quan Zongliang)
- 添加Windows上支持收集 崩溃转储 (Craig Ringer, Magnus Hagander)

  通过非调试Windows二进制文件产生minidumps， 并且通过标准调试工具进行分析。

- 启动MinGW64编译器编译(Andrew Dunstan)

  这允许编译64位Windows二进制文件即使在非Windows平台通过交叉编译。

E.19.3.12. 源代码

- 修订GUC变量分配钩的API(Tom Lane)

以往分配钩函数划分检查钩和分配钩， 前者可以失败但是后者不能。 这一变化将影响定义自定义GUC参数的附加模块。

- 添加源代码锁存器以支持等待事件(Heikki Linnakangas)
- 集中数据修改权限检查逻辑(KaiGai Kohei)
- 为了保持一致性添加缺失的 `get_` *object* `_oid()` 函数(Robert Haas)
- 为了compiling add-on modules 通过删除冲突关键字提高使用C++编译器的能力(Tom Lane)
- 添加支持DragonFly BSD (Rumko)
- 出于后端使用暴露 `quote_literal_cstr()` (Robert Haas)
- 在缺省编码中运行回归测试(Peter Eisentraut)

  回归测试以前总是运行 `SQL_ASCII` 编码。
- 添加src/tools/git_changelog替换 cvs2cl和pgcvslog (Robert Haas, Tom Lane)
- 添加git-external-diff脚本到 `src/tools` (Bruce Momjian)

  这用于产生来自git的语境差异。
- 提高支持编译Clang (Peter Eisentraut)

E.19.3.12.1. 服务器钩

- 添加源代码钩检查权限(Robert Haas，Stephen Frost)
- 出于使用安全框架添加后对象创建函数钩(KaiGai Kohei)
- 添加客户端认证钩(KaiGai Kohei)

E.19.3.13. Contrib

- 修改 `contrib` 模块和程序语言通过新的 扩展 机制安装(Tom Lane, Dimitri Fontaine)
- 添加 `contrib/file_fdw` 外数据包(Shigeru Hanada)

  外表使用这个外数据包可以以类似于 `COPY` 的方式读取平文件。
- 添加最近邻搜索支持 `contrib/pg_trgm` 和 `contrib/btree_gist` (Teodor Sigaev)
- 添加 `contrib/btree_gist` 支持不平等搜索(Jeff Davis)
- 修复 `contrib/fuzzystrmatch` 的 `levenshtein()` 函数以处理多字节字符(Alexander Korotkov)

- 添加 `ssl_cipher()` and `ssl_version()` 函数到 `contrib/sslinfo` (Robert Haas)
- 修复 `contrib/intarray` 和 `contrib/hstore` 提供与索引空数组一致结果(Tom Lane)

  先前使用索引的空数组查询可能返回使用顺序扫描的不同结果。
- 允许 `contrib/intarray` 恰当工作在多维数组上(Tom Lane)
- 在 `contrib/intarray` 上， 避免在没有空值实际存在的情况下抱怨空值存在的错误(Tom Lane)
- 在 `contrib/intarray` 中， 修复关于空数组包含操作符操作(Tom Lane)

  空数组现在正确地认为被包含在任何其他数组中。
- 删除通过 `xslt_process()` 处理的 `_parameter_` = `_value_` 对数上 `contrib/xml2` 的任意限制(Pavel Stehule)

  以前限制是10。
- 在 `contrib/pageinspect` 中， 修复heap_page_item以返回32位值的infomask(Alvaro Herrera)

  这避免返回负值， 这是混乱的。 该潜在值是16位无符号整数。

E.19.3.13.1. 安全

- 添加 `contrib/sepgsql` 连接与SELinux的权限检查(KaiGai Kohei)

  这使用新的 `SECURITY LABEL` 功能。
- 添加contrib模块 `auth_delay` (KaiGai Kohei)

  在返回认证失败之前导致服务器暂停； 设计它使得蛮力密码攻击更加困难。
- 添加 `dummy_seclabel` contrib模块(KaiGai Kohei)

  这用于允许回归测试。

E.19.3.13.2. 性能

- 添加支持 `LIKE` and `ILIKE` 索引搜索到 `contrib/pg_trgm` (Alexander Korotkov)
- 添加 `levenshtein_less_equal()` 函数到 `contrib/fuzzystrmatch` ， 这是对小距离的优化(Alexander Korotkov)
- 提高 `contrib/seg` 上索引查找性能 (Alexander Korotkov)
- 提高许多关系数据库 pg_upgrade 的性能(Bruce Momjian)
- 添加标记到 `contrib/pgbench` 以报告每个语句延迟(Florian Pflug)

E.19.3.13.3. Fsync测试

- 移动 `src/tools/test_fsync` 到 `contrib/pg_test_fsync` (Bruce Momjian, Tom Lane)
- 添加 `O_DIRECT` 支持到 `contrib/pg_test_fsync` (Bruce Momjian)

  这匹配通过 `wal_sync_method` 的 `O_DIRECT` 的使用。

- 添加新的测试给 `contrib/pg_test_fsync` (Bruce Momjian)

E.19.3.14. 文档

- 广泛的ECPG 文档改进(Satoshi Nagayasu)
- 广泛的校对和文档改进 (Thom Brown, Josh Kupershmidt, Susanne Ebrecht)
- 为 `exit_on_error` 添加文档(Robert Haas)

  这个参数导致会话有任何错误就退出。

- 为 `pg_options_to_table()` 添加文档(Josh Berkus)

  这个函数显示了可读形式中表存储选项。

- 文档可以访问使用 `(compositeval).*` 语法的所有复合类型字段(Peter Eisentraut)
- 文档 `translate()` 删除没有相应的 `to` 字符的 `from` 中字符(Josh Kupershmidt)
- 为 `CREATE CONSTRAINT TRIGGER` 和 `CREATE TRIGGER` 合并文档(Alvaro Herrera)
- 集中权限和 升级文档(Bruce Momjian)
- 为Solaris 10添加内核调整记录 (Josh Berkus)

  先前只有Solaris 9 内核调整被记录。

- 一致地处理 `HISTORY` 文件中的非ASCII字符(Peter Eisentraut)

  当 `HISTORY` 文件是英语， 我们必须处理参与者名字中非ASCII字母。 这些目前是音译的， 因此它们没有关于字符集假设相当易读。

E.20. 版本 9.0.14

> 发布日期: 2013-10-10

这个版本包含各种自9.0.13以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.20.1. 迁移到版本 9.0.14

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.20.2. 修改列表

- 阻止多字节编码中非ASCII非双引号的标识符的小写转换 (Andrew Dunstan)

  先前的行为是错误并且混乱的。

- 修复 `wal_level = hot_standby` 时后台写作检查点内存泄露 (Naoya Anzai)

- 修复 `lo_open()` 失败导致的内存泄露 (Heikki Linnakangas)

- 修复 `work_mem` 使用超过24GB内存的内存过度使用错误 (Stephen Frost)

- 修复libpq SSL死锁错误 (Stephen Frost)

- 修复线程libpq应用程序中可能的SSL网络堆栈损坏 (Nick Phillips, Stephen Frost)

- 适当的计算包含许多NULL值的布尔字段的行估计 (Andrew Gierth)

  以前的文本，像 `col IS NOT TRUE` 和 `col IS NOT FALSE` ， 在估计规划开销时并不能适当的包括进NULL值中。

- 阻止将 `WHERE` 子句下推到不安全的 `UNION/INTERSECT` 子查询中 (Tom Lane)

  先前这样的下推可能产生错误。

- 修复不适当的处理数据类型修改引起的稀有的 `GROUP BY` 查询错误 (Tom Lane)

- 允许视图转储代码在基表上更好的处理已删除的字段 (Tom Lane)

- 适当的记录用 `UNIQUE` 和 `PRIMARY KEY` 语法创建的索引注释 (Andres Freund)

  这修复了一个并行的pg_restore错误。

- 修复了 `REINDEX TABLE` 和 `REINDEX DATABASE`， 以正确的使约束重新生效，并且标记无效的索引为有效 (Noah Misch)

  `REINDEX INDEX` 总是正常工作。

- 修复并发 `CREATE INDEX CONCURRENTLY` 操作期间可能的死锁 (Tom Lane)

- 修复 `regexp_matches()` 处理零长度匹配 (Jeevan Chalke)

  以前，像'^'这样的零长度匹配可能返回很多匹配。

- 修复过度复杂的正则表达式的崩溃 (Heikki Linnakangas)

- 修复正则表达式逆向引用和非贪婪量词结合的匹配错误 (Jeevan Chalke)

- 阻止 `CREATE FUNCTION` 检查 `SET` 变量，除非启用了函数体检查 (Tom Lane)

- 允许 `ALTER DEFAULT PRIVILEGES` 在模式上操作，不需要CREATE权限 (Tom Lane)

- 放松了在查询上使用的关键字的限制 (Tom Lane)

  特别的，减少了角色名、语言名、 `EXPLAIN` 和 `COPY` 操作、 还有 `SET` 值的关键字的限制。这允许 `COPY ... (FORMAT BINARY)` 以前的 `BINARY` 需要单引号。

- 修复了 `pgp_pub_decrypt()`， 这样它为带有口令的秘钥工作 (Marko Kreen)

- 删除缺少索引的表vacuum期间稀有的不准确的警告 (Heikki Linnakangas)

- 改善取消文件截断请求之后的分析统计的生成 (Kevin Grittner)

- 避免在预备查询中执行事务控制命令时可能的错误（如 ROLLBACK） (Tom Lane)

- 允许在所有平台上无穷的各种拼写 (Tom Lane)

  支持的无穷的值是"inf", "+inf", "-inf", "infinity", "+infinity", 和 "-infinity"。

- 扩张比较行的能力到记录和数组 (Rafal Rzepecki, Tom Lane)

- 更新时区数据文件到tzdata版本2013d，因为DST规律在Israel, Morocco, Palestine, Paraguay方面改变了。另外，Macquarie Island历史时区数据纠正 (Tom Lane)

E.21. 版本 9.0.13

> 发布日期: 2013-04-04

这个版本包含各种自9.0.12以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.21.1. 迁移到版本 9.0.13

运行9.0.X的系统不需要转储/恢复。

不过，这个版本纠正了几个管理GiST索引的错误。在安装这个更新之后， 建议 `REINDEX` 任意满足一个或多个下面描述的条件的GiST索引。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.21.2. 修改列表

- 修复了不安全的服务器命令行开关的分析 (Mitsumasa Kondo, Kyotaro Horiguchi)

  一个包含数据库名的连接请求以" `-` "开始有可能被破坏， 或破坏服务器的数据目录中的文件，即使最终拒绝了该请求。 (CVE-2013-1899)

- 在每个postmaster子进程中重置OpenSSL随机状态 (Marko Kreen)

  这避免了 `contrib/pgcrypto` 函数生成的随机数很容易被另一个数据库用户猜到的情况。 该风险只在postmaster用 `ssl` = `on` 配置但是大多数连接不使用SSL加密时是重要的。 (CVE-2013-1900)

- 修复了GiST索引在不适当的时候不使用"fuzzy"几何比较 (Alexander Korotkov)

  核心几何类型执行比较使用"fuzzy"相等，但是 `gist_box_same` 必须做精确的比较，否则GiST使用它索引可能变成不一致的。在安装这个更新之后， 用户应该 `REINDEX` 在 `box` , `polygon` , `circle` , 或 `point` 字段之上的任何GiST索引，因为所有这些索引都使用 `gist_box_same` 。

- 修复为变量范围的数据类型使用 `contrib/btree_gist` 的错误的范围联合和GiST索引中的处罚逻辑，这些是 `text` , `bytea` , `bit` , 和 `numeric` 字段 (Tom Lane)

  这些错误可能导致不一致的索引，使某些现有的键不能被搜索到，和无用的索引膨胀。 建议用户在安装这个更新之后 `REINDEX` 这样的索引。

- 修复多字段索引的GiST页分裂代码中的错误 (Tom Lane)

这些错误可能导致不一致的索引，使某些现有的键不能被搜索到，和索引中不需要的无效搜索。 建议用户在安装这个更新之后 `REINDEX` 多字段GiST索引。

- 修复 `gist_point_consistent` 处理模糊一致 (Alexander Korotkov)

  Gist索引在 `point` 字段上的索引扫描可能有时产生不同于顺序扫描的结果， 因为 `gist_point_consistent` 与潜在的操作符代码关于是否做比较或模糊不一致。

- 修复WAL重放中的缓冲区泄露 (Heikki Linnakangas)

  这个错误可能导致重放期间的"incorrect local pin count"错误，使恢复成为不可能。

- 修复 `DELETE RETURNING` 中的竞态条件 (Tom Lane)

  在正确的情况下， `DELETE RETURNING` 可以尝试从当前进程不再有任何pin的共享缓冲区获取数据。 如果同时其他的进程改变了该缓冲区，那么会导致垃圾 `RETURNING` 输出， 或者甚至会崩溃。

- 修复正则表达式编译中无限循环的风险 (Tom Lane, Don Porter)

- 修复正则表达式编译中潜在的空指针解除参照 (Tom Lane)

- 修复 `to_char()` 在适当的地方只使用ASCII小写化规则 (Tom Lane)

  这个修复了一些应该环境独立的临时模式的错误行为，但是在Turkish环境中错误操作" `I` "和" `i` "。

- 修复了不想要的时间戳 `1999-12-31 24:00:00` 的拒绝

- 修复了一个单一事务做 `UNLISTEN` 然后 `LISTEN` 的逻辑错误 (Tom Lane)

  该会话一点也不监听通知事件，尽管它在这种情况下很确定的应该监听。

- 删除没有用的"picksplit doesn't support secondary split"日志消息 (Josh Hansen, Tom Lane)

  这个消息看起来已经添加到了从没有写过的期望代码，并且可能永远不写， 因为GiST的二次分裂的缺省处理实际上非常好。所以停止给最终用户说这个。

- 修复可能的错误：发送会话的最后几个事务提交/终止计数到统计收集器 (Tom Lane)

- 消除PL/Perl的 `spi_prepare()` 函数中的内存泄露 (Alex Hunsaker, Tom Lane)

- 修复pg_dumpall处理正确的包含" `=` "的数据库名 (Heikki Linnakangas)

- 当给出不正确的连接字符串时避免pg_dump中的崩溃 (Heikki Linnakangas)

- 忽略pg_dump和pg_upgrade中无效的索引 (Michael Paquier, Bruce Momjian)

转储无效的索引会在恢复时导致问题，例如， 如果索引创建失败的原因是因为它试图通过表的数据强制一个不合适的唯一条件。 还有，如果索引创建实际上仍在进行，那么认为这是一个未提交的DDL变化看起来是合理的， pg_dump无论如何也不会转储。 pg_upgrade 现在也跳过无效的索引而不是失败。

- 修复 `contrib/pg_trgm` 的 `similarity()` 函数， 使其为trigram-less字符串返回0 (Tom Lane)

  以前因为内部除零，它返回 `NaN` 。

- 更新时区数据文件为tzdata版本2013b，为DST规律在Chile, Haiti, Morocco, Paraguay, 和一些Russian地区的改变。还有，为许多地区修正历史的时区数据。

  还有，为在Russia和其他地方里的最近的改变更新时区缩写文件： `CHOT` , `GET` , `IRKT` , `KGT` , `KRAT` , `MAGT` , `MAWT` , `MSK` , `NOVT` , `OMST` , `TKT` , `VLAT` , `WST` , `YAKT` , `YEKT` 现在跟随它们的当前含义， `VOLT` (Europe/Volgograd)和 `MIST` (Antarctica/Macquarie)添加到了缺省的缩写列表。

E.22. 版本 9.0.12

> 发布日期: 2013-02-07

这个版本包含各种自9.0.11以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.22.1. 迁移到版本 9.0.12

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.22.2. 修改列表

- 阻止SQL执行 `enum_recv` (Tom Lane)

  该函数是错误声明的，允许一个简单的SQL命令使服务器崩溃。原则上， 攻击者可以使用它检验服务器内存的内容。感谢Sumit Soni (via Secunia SVCRP) 报告这个问题。(CVE-2013-0255)

- 修复在WAL回放期间已经达到一个一致的数据库状态时检测的多重问题 (Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- 截断一个关系文件时更新最小的恢复点 (Heikki Linnakangas)

  一旦数据被丢弃，在时间线中的一个较早的点停止恢复已经不再安全。

- 修复热备份模式中丢失的取消 (Noah Misch, Simon Riggs)

  有时会错过取消冲突的热备份查询的需要，此时允许这些查询看到不一致的数据。

- 修复SQL语法允许一个子SELECT结果加下标或字段选择 (Tom Lane)

- 修复繁忙的工作负载中自动清理截断的性能问题 (Jan Wieck)

  在一个表的最后截断空白页需要排他锁，但是当有冲突的锁请求时autovacuum编码失败（并释放表锁）。 负载过轻，很有可能不会发生截断，导致表膨胀。通过执行一个部分的截断，释放该锁， 然后尝试再次请求这个锁并继续来修复这个问题。这个修复也大大的减少了autovacuum 释放该锁之前冲突的锁请求到达之后的平均时间。

- 扫描 `pg_tablespace` 时防止竞态条件 (Stephen Frost, Tom Lane)

如果 `CREATE DATABASE` 和 `DROP DATABASE` 并发的更新 `pg_tablespace` 记录，那么它们可能行为错误。

- 阻止 `DROP OWNED` 试图删除整个数据库或表空间 (Álvaro Herrera)

  为了安全，这些对象的所有权必须重新分配，而不是抛弃。

- 修复 `vacuum_freeze_table_age` 实现中的错误 (Andres Freund)

  在有多于 `vacuum_freeze_min_age` 个事务存在的安装中，这个错误阻止使用部分表扫描的autovacuum，所以全表扫描将总是发生。

- 阻止 `RowExpr` 或 `XmlExpr` 解析分析两次时的错误行为 (Andres Freund, Tom Lane)

  这个错误在内容中可能是用户可见的，如 `CREATE TABLE LIKE INCLUDING INDEXES` 。

- 提高哈希表大小计算中的整数溢出防御 (Jeff Davis)

- 拒绝 `to_date()` 中超出范围的日期 (Hitoshi Harada)

- 确保非ASCII提示字符串在Windows上转化为正确的代码页 (Alexander Law, Noah Misch)

  这个错误影响psql和一些其他客户端程序。

- 修复没有连接到一个数据库时psql的 `\?` 命令中可能的崩溃 (Meng Qingzhong)

- 修复pg_upgrade安全的处理无效的索引 (Bruce Momjian)

- 修复libpq的 `PQprintTuples` 中的一个字节缓冲区溢出 (Xi Wang)

  这个古老的函数PostgreSQL本身已经不用了，但是一些客户端代码可能仍然在使用。

- 使ecpglib适当的使用翻译了的信息 (Chen Huajun)

- 在MSVC上适当的安装ecpg_compat和pgtypes (Jiang Guiqing)

- 在libecpg中包括我们的 `isinf()` 版本， 如果系统没有提供它 (Jiang Guiqing)

- 为提供的函数重新排列配置的测试，这样它不会被来自libedit/libreadline的虚假报告欺骗 (Christoph Berg)

- 确保Windows建立编号在时间上增加 (Magnus Hagander)

- 当在Windows上交叉编译时，使带有正确 `.exe` 后缀的 pgxs建立可执行 (Zoltan Boszormenyi)

- 添加新的时间戳缩写 `FET` (Tom Lane)

  这个现在用于一些东欧时区。

E.23. 版本 9.0.11

> 发布日期: 2012-12-06

这个版本包含各种自9.0.10以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.23.1. 迁移到版本 9.0.11

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.23.2. 修改列表

- 修复了多个与 `CREATE INDEX CONCURRENTLY` 相关的错误 (Andres Freund, Tom Lane)

  修复 `CREATE INDEX CONCURRENTLY` 以在改变一个索引的 `pg_index` 行的状态时使用按照地点的更新。这阻止了会导致并发会话错过更新目标索引的竞态条件， 因此导致损坏并发创建的索引。

  另外，修复了各种其他操作，以保证它们忽略失败的 `CREATE INDEX CONCURRENTLY` 命令产生的无效的索引。最重要的是 `VACUUM`， 因为在采取正确的动作修复或删除无效的索引之前，一个auto-vacuum可以很容易的在一个表上发起。

- 修复WAL重放期间的缓冲区锁定 (Tom Lane)

  当重放WAL记录影响多于一页时，WAL重放代码关于锁定缓冲区不够小心。 这会导致热备份查询短暂的看到不一致的状态，导致错误响应或意外的失败。

- 修复在GIN索引的WAL生产逻辑中的一个错误 (Tom Lane)

  如果发生页撕裂错误，这会导致索引损坏。

- 当推进热备份服务器正常运行时，适当的删除启动进程的虚拟XID锁 (Simon Riggs)

  这个监管可以阻止特定操作（如 `CREATE INDEX CONCURRENTLY` ）的连续执行。

- 避免备用模式中虚假的"out-of-sequence timeline ID"错误 (Heikki Linnakangas)

- 阻止postmaster在收到关闭信号后发起新的子进程 (Tom Lane)

  这个错误能引起关闭的时间更长，甚至在没有额外的用户动作下永不完成。

- 避免内存溢出时内部哈希表的损坏 (Hitoshi Harada)
- 修复外连接上非绝对相等子句的规划 (Tom Lane)

  该规划会导致不正确的约束：一个子句等于一个其他非绝对构造，例如 `WHERE COALESCE(foo, 0) = 0`，当 `foo` 来自一个外连接的可为null方时。

- 改善规划的能力，检验来自相等子句的排他约束 (Tom Lane)
- 修复散列的子规划中的部分行匹配，正确的处理交叉类型情况 (Tom Lane)

  这个影响多字段 `NOT IN` 子规划，如 `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` 中 `b` 和 `y` 分别是 `int4` 和 `int8` 的情况。 这个错误导致错误响应或崩溃依赖于特定包含的数据类型。

- 当为一个 `AFTER ROW UPDATE/DELETE` 触发器重新抓取老的元组时获得缓冲区锁 (Andres Freund)

  在不寻常的情况下，这个疏忽会导致外键强制触发器传送不正确的数据到预先检查逻辑。 这样会导致一个崩溃，或一个关于是否触发该触发器的不正确的决定。

- 修复 `ALTER COLUMN TYPE` 正确的处理非继承的检查约束 (Pavan Deolasee)

  这在8.4之前的版本中运行正确，现在在8.4和之后的版本中也正确运行了。

- 修复 `REASSIGN OWNED` 处理表空间上的授权 (Álvaro Herrera)
- 为视图系统字段忽略不正确的 `pg_attribute` 条目 (Tom Lane)

  视图没有任何系统字段。不过，我们在转换一个表到一个视图时忘记删除这样的条目。 这在9.3和之后的版本中适当的修复了，但是，在之前的分支中我们需要防卫现存的错误转换的视图。

- 修复规则打印以正确的转储 `INSERT INTO` `_table_` DEFAULT VALUES (Tom Lane)
- 当在一个查询中有太多 `UNION` / `INTERSECT` / `EXCEPT` 子句时，防止堆栈溢出 (Tom Lane)
- 当用-1除最小可能的整数时，阻止依赖于平台的失败 (Xi Wang, Tom Lane)
- 修复日期解析中可能的以字符串结束的存取经过 (Hitoshi Harada)
- 修复在检查点和 `wal_level` 是 `hot_standby` 期间， 如果发生XID打包未能提前XID纪元的错误 (Tom Lane, Andres Freund)

  这个错误对PostgreSQL本身没有特别的影响， 不利于依赖于 `txid_current()` 和相关函数的应用： TXID值将会看起来向后退了。

- 如果一个Unix域套接字的路径名长度超出平台特定的限制，则产生一个可以理解的错误信息 (Tom Lane, Andrew Dunstan)

以前，这会产生相当没有帮助的事物，如"Non-recoverable failure in name resolution"。

- 修复发送复合字段值到客户端时的内存泄露 (Tom Lane)
- 使pg_ctl读取 `postmaster.pid` 文件更强健 (Heikki Linnakangas)

  修复竞态条件和可能的文件描述符泄露。

- 修复psql中可能的崩溃，如果提出了不正确编码的数据和 `client_encoding` 设置是一个客户端唯一编码（如 SJIS） (Jiang Guiqing)
- 修复 `restore.sql` 脚本中的错误，该脚本由pg_dump 以 `tar` 输出格式发出 (Tom Lane)

  该脚本在名字包含大写字母的表上将会完全的失败。另外， 使得该脚本可以在 `--inserts` 模式和定期的COPY模式中存储数据。

- 修复pg_restore接受符合POSIX的 `tar` 文件 (Brian Weaver, Tom Lane)

  pg_dump的 `tar` 输出模式的原始编码产生的文件并不完全符合POSIX标准。 这在版本9.3中已经纠正了。这个补丁更新之前的分支，所以它们将接受正确和不正确的格式， 希望9.3出来时避免兼容性问题。

- 修复pg_resetxlog以正确的定位 `postmaster.pid` ， 当给出一个相关的路径到数据目录时 (Tom Lane)

  这个错误会导致pg_resetxlog不注意这里有一个活动的postmaster使用该数据目录。

- 修复libpq的 `lo_import()` 和 `lo_export()` 函数以适当的报告文件I/O错误 (Tom Lane)
- 修复ecpg嵌套结构指针变量的流程 (Muhammad Usama)
- 修复ecpg的 `ecpg_get_data` 函数正确的处理数组 (Michael Meskes)
- 使 `contrib/pageinspect` 的btree页面检查函数在检查页面时获得缓冲区锁 (Tom Lane)
- 修复pgxs支持以在AIX上建立可加载的模块 (Tom Lane)

  在不在AIX上运行的原始源代码树之外建立模块。

- 更新时区数据文件到tzdata版本2012j，因为DST规律在下列地区改变了： Cuba, Israel, Jordan, Libya, Palestine, Western Samoa, 还有部分 Brazil。

E.24. 版本 9.0.10

> 发布日期: 2012-09-24

这个版本包含各种自9.0.9以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.24.1. 迁移到版本 9.0.10

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.24.2. 修改列表

- 修复规划者的执行者参数的分配，修复执行者为CTE规划节点的重新扫描逻辑 (Tom Lane)

  这些错误会导致查询的错误响应，使得扫描多次相同的 `WITH` 子查询。

- 改善GiST索引中的页面分裂决策 (Alexander Korotkov, Robert Haas, Tom Lane)

  由于这个错误，多字段GiST索引可能遭受意外的膨胀。

- 如果仍然持有权限，修复级联权限撤销停止 (Tom Lane)

  如果我们从一些角色 _X_ 撤销授予选项，但是 _X_ 通过来自其他的授予仍然持有这个选项，我们不应该递归的从 _X_ 授予的角色 _Y_ 撤销相应的权限。

- 改善热备份错误配置错误的错误消息 (Gurjeet Singh)

- 修复使用PL/Perl时的 `SIGFPE` 的处理 (Andres Freund)

  Perl重置进程的 `SIGFPE` 处理器为 `SIG_IGN`， 这个稍后会导致崩溃。在初始化PL/Perl之后恢复正常的Postgres信号处理器。

- 如果递归的PL/Perl函数在执行时被重新定义，那么阻止PL/Perl崩溃 (Tom Lane)

- 绕开PL/Perl中可能的错误最优化 (Tom Lane)

  一些Linux发布包含一个不正确的 `pthread.h` 版本， 导致PL/Perl中不正确的编译代码，如果PL/Perl函数调用另一个抛出一个错误的函数，会导致崩溃。

- 修复Windows上pg_upgrade处理行尾结束符 (Andrew Dunstan)

以前，pg_upgrade可能添加或删除回车的地方如函数体。

- 在Windows上，pg_upgrade在它发出的脚本里使用反斜杠路径分隔符 (Andrew Dunstan)
- 更新时区数据文件到tzdata版本2012f，因为DST规律在Fiji改变了。

E.25. 版本 9.0.9

> 发布日期: 2012-08-17

这个版本包含各种自9.0.8以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.25.1. 迁移到版本 9.0.9

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.25.2. 修改列表

- 阻止通过XML实体引用访问外部文件/URL (Noah Misch, Tom Lane)

  `xml_parse()` 将尝试获取解决在XM值中引用的DTD和实体所需要的外部文件或URL， 因此允许非特权数据库用户使用数据库服务器的权限尝试获取数据。 但是外部数据不会直接返回给用户，如果该数据不可以解析为合法的XML， 则它的一部分会以错误消息的方式外露；并且在任何情况下， 仅仅能够检查文件的存在可能对一个攻击者有用。(CVE-2012-3489)

- 阻止通过 `contrib/xml2` 的 `xslt_process()` 访问外部文件/URL (Peter Eisentraut)

  libxslt提供通过样式表命令读写文件和URL的能力， 因此允许非特权的数据库用户使用数据库服务器的权限读写数据。 通过适当的使用libxslt的安全选项禁用该功能。 (CVE-2012-3488)

  另外，删除 `xslt_process()` 从外部文件/URL获取文档和样式表的能力。 虽然这是记录中的"特性"，但是它长期被认为是一个坏的想法。 CVE-2012-3489修复打破了这个能力， 并且与其花费时间尝试修复它， 不如立刻删除它。

- 阻止btree索引页过早的回收 (Noah Misch)

  当我们允许只读事务跳过设定XID时，我们引入了已删除的btree索引页可以被回收的可能性， 而一个只读事务仍然在该索引页中运行。这可能会导致不正确的索引搜索结果。 这样一个错误发生在字段中的可能性因为计时要求看起来非常低，但是，尽管如此也应该修复它。

- 用新创建的或重新设置的序列修复崩溃安全bug (Tom Lane)

如果 `ALTER SEQUENCE` 在一个新创建的或重置的序列上执行， 并且正好在它上面有一个 `nextval()` 调用，那么然后服务器崩溃了， WAL重放将恢复该序列到 `nextval()` 还没有做的那个状态， 因此允许第一个序列值再次被下一个 `nextval()` 调用返回。 特别的，这会为 `serial` 列显示，因为串行列序列的创建包括一个 `ALTER SEQUENCE OWNED BY` 步骤。

- 修复 `txid_current()` ，当不是在热备时报告正确的纪元 (Heikki Linnakangas)

  这个修复了前一个小版本中引入的回归。

- 修复热备启动里的bug，当一个主要事务有许多子事务时 (Andres Freund)

  这个错误导致和"out-of-order XID insertion in KnownAssignedXids"一样的失败报告。

- 确保 `backup_label` 文件在 `pg_start_backup()` 之后是同步的 (Dave Kerr)

- 修复walsender进程中的超时处理 (Tom Lane)

  WAL发送后端进程忽略了建立一个SIGALRM处理器， 意味着它们在一些极端情况下将一直等待，而这种情况应该发生的是超时。

- 改善Back-patch 9.1以压缩同步请求队列 (Robert Haas)

  这提高了检查点期间的性能。该9.1改变现在看起来对于back-patch字段测试足够安全。

- 修复 `LISTEN` / `NOTIFY` 以更好的处理I/O问题， 例如超出磁盘空间 (Tom Lane)

  在写入失败之后，所有随后发送更多 `NOTIFY` 信息的尝试都将失败， 带有像这样的信息："Could not read from file "pg*notify/`_nnnn* `" at offset` *nnnnn*`: Success"。

- 只允许autovacuum通过直接阻塞进程被自动取消 (Tom Lane)

  原始编码允许在某些情况下不一致的行为；特别的，autovacuum在少于 `deadlock_timeout` 宽限期后被取消。

- 改善autovacuum取消的登陆 (Robert Haas)

- 修复日志收集器，以便 `log_truncate_on_rotation` 在 服务器启动后的第一个日志旋转期间工作 (Tom Lane)

- 修复 `WITH` 附属于一个嵌套的集合运算 (`UNION` / `INTERSECT` / `EXCEPT`) (Tom Lane)

- 确保整个行引用子查询不会包括任何额外的 `GROUP BY` 或 `ORDER BY` 列 (Tom Lane)

- 不允许在 `CREATE TABLE` 期间在 `CHECK` 约束和索引定义中拷贝整个行引用 (Tom Lane)

  这种情况会出现在带有 `LIKE` 或 `INHERITS` 的 `CREATE TABLE` 中。 拷贝的整个行变量用原始表而不是新表的行类型错误的标记。拒绝 `LIKE` 看起来合理的情况，因为行类型可能稍后会有分歧。对于 `INHERITS` ， 我们应该允许它，隐式转换为当前表的行类型；但是这样

将请求更多的工作。

- 修复 `ARRAY(SELECT ...)` 子查询中的内存泄露(Heikki Linnakangas, Tom Lane)
- 修复正则表达式中公共前缀的提取 (Tom Lane)

  该代码会对量化的加上括号的子表达式感到困惑，如 `^(foo)?bar` 。 这会导致对这种模式的搜索的不正确的索引优化。

- 修复 `interval` 常量中有符号的 `_hh_``:``_mm_` 和 `_hh_``:``_mm_``:``_ss_` 字段分析的bug (Amit Kapila, Tom Lane)
- 当在PL/Python中转换一个Python Unicode字符串为服务器编码时， 使用Postgres的编码转换函数，而不是Python的 (Jan Urbanski)

  这避免了一些极端情况问题，尤其是Python不支持所有的Postgres编码。 一个值得注意的功能性改变是，如果服务器编码是SQL_ASCII， 你将得到该字符串的UTF-8表示；以前，字符串中的任何非ASCII字符都将导致一个错误。

- 修复PL/Python中PostgreSQL编码的映射为Python编码 (Jan Urbanski)
- 适当的报告 `contrib/xml2` 的 `xslt_process()` 中的错误 (Tom Lane)
- 更新时区数据文件为tzdata版本2012e，因为Morocco和Tokelau中的DST规律改变。

E.26. 版本 9.0.8

> 发布日期: 2012-06-04

这个版本包含自9.0.7以来的各种修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.26.1. 迁移到版本 9.0.8

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.26.2. 修改列表

- 修复 `contrib/pgcrypto` 的 DES `crypt()` 函数中不正确的口令转换 (Solar Designer)

  如果一个口令字符串包含字节值 `0x80`， 则忽略剩余的口令， 导致口令比它表现出来的还要弱。有了这个修复，剩余的字符串适当的包含在DES散列中。 这个bug影响的任何存储的口令值将因此而不再匹配，所以存储的值可能需要更新。 (CVE-2012-2143)

- 为过程语言的调用处理器忽略 `SECURITY DEFINER` 和 `SET` 属性 (Tom Lane)

  应用这样的属性到一个调用处理器会使服务器崩溃。 (CVE-2012-2655)

- 允许 `timestamp` 输入中的数字时区偏移量距离UTC多达16个小时 (Tom Lane)

  一些历史时区的偏移量大于15个小时，这是以前的限制。这会导致转储的数据值在重载时被拒绝。

- 修复给定时间对于当前时区来说正好是最后的DST转换时间时的时间戳转换处理 (Tom Lane)

  这个监管已经有很长一段时间了，但是之前没有注意到是因为大多数使用DST的时区，假设有一个未来DST转换的不确定序列。

- 修复 `text` 到 `name` 和 `char` 到 `name` 的转换， 以在多字节编码中正确的执行字符串截断 (Karl Schnaitter)

- 修复 `to_tsquery()` 中的内存复制bug (Heikki Linnakangas)

- 确保在热备上执行时 `txid_current()` 报告正确的纪元 (Simon Riggs)

- 修复规划师处理子查询中外部PlaceHolderVars (Tom Lane)

这个bug关注引用变量的子SELECT，该变量来自周围查询的外连接的可以为空侧。 在9.1中，被这个bug影响的查询将会失败："ERROR: Upper-level PlaceHolderVar found where not expected"。 但是在9.0和8.4中，你可能只是得到错误的响应，因为传送到子查询中的值在应该为空时不是空。

- 修复`pg_attribute`非常大时会话启动缓慢 (Tom Lane)

  如果`pg_attribute`超过`shared_buffers`的四分之一， 在会话启动将触发同步的扫描逻辑时，有时会需要缓存重建代码， 导致花费的时间比平时要多一些。如果一次启动许多新会话，这个问题尤其严重。

- 确保适当频度的取消查询的序列化扫描检查 (Merlin Moncure)

  一个扫描遇到许多包含非活动元组的页面时将不会响应中断。

- 确保Windows实现在返回前`PGSemaphoreLock()`清理`ImmediateInterruptOK` (Tom Lane)

  这个疏忽意味着在一个不安全的时间查询取消中断的接收将晚于相同查询的接受， 会有不可预料但不好的后果。

- 打印视图或规则时安全的显示整行变量 (Abbas Butt, Tom Lane)

  极端情况下包括歧义的名字（也就是，该名字是该查询的表名或者字段名）会以一个歧义的方式输出， 导致视图或规则在转储或重载时有不同解释的风险。通过附加一个no-op转换避免歧义情况。

- 修复`COPY FROM`以正确的处理相当于无效编码的空标记字符串 (Tom Lane)

  空标记字符串如`E'\\0'`应该生效，并且在过去确实生效，但是在8.4中这种情况就打破了。

- 确保autovacuum工作进程正确的执行堆栈深度检查 (Heikki Linnakangas)

  以前，自动`ANALYZE`在一个函数中调用无限递归会使工作进程崩溃。

- 修复日志收集器，使其在高度负载下不会丢失日志一致性 (Andrew Dunstan)

  该收集器在以前如果太忙碌会在重新装配大的信息时失败。

- 修复日志收集器，确保在接收到SIGHUP后重启文件循环 (Tom Lane)

- 修复GIN索引的WAL重放逻辑，如果索引稍后删除则使其不会失败 (Tom Lane)

- 修复PL/pgSQL的`RETURN NEXT`命令中的内存泄露 (Joe Conway)

- 修复PL/pgSQL的`GET DIAGNOSTICS`命令，当目标是函数的第一个变量时 (Tom Lane)

- 修复psql的扩展显示(`\x`)模式中潜在的内存访问结束 (Peter Eisentraut)

- 修复数据库包含许多对象时pg_dump中的几个性能问题 (Jeff Janes, Tom Lane)

如果数据库包含许多模式，或者如果许多对象在依赖循环中，或者如果有许多自身拥有的序列， 那么pg_dump会变得非常缓慢。

- 修复pg_upgrade，以防数据库以非缺省的表空间存储， 而该表空间包含一个在集群的缺省表空间的表的情况 (Bruce Momjian)
- 在ecpg中，修复少见的内存泄露和可能的 `sqlca_t` 结构之后的一个字节的重写 (Peter Eisentraut)
- 修复 `contrib/dblink` 的 `dblink_exec()` 以在错误时不泄露临时数据库连接 (Tom Lane)
- 修复 `contrib/dblink` 以在错误信息中报告正确的连接名 (Kyotaro Horiguchi)
- 修复 `contrib/vacuumlo` 以在删除许多大对象时使用多个事物 (Tim Lewis, Robert Haas, Tom Lane)

  这个改变在有许多大对象需要删除时，避免了过度的 `max_locks_per_transaction` 。 该行为可以根据新增的 `-l` （limit）选项调整。

- 更新时区数据文件到tzdata版本2012c，因为DST规律在 Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria, 和 Tokelau Islands改变了；还有为加拿大的历史的更正。

E.27. 版本 9.0.7

> 发布日期: 2012-02-27

这个版本包含各种自9.0.6以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.27.1. 迁移到版本 9.0.7

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的版本说明。

E.27.2. 修改列表

- 需要在触发器函数`CREATE TRIGGER`上的执行权限 (Robert Haas)

  这个缺失的检查会允许另外一个用户通过在他自己的一个表上安装触发器来用伪造的输入数据执行触发器函数。 这只在触发器函数标记为`SECURITY DEFINER`时是有意义的， 因为否则触发器函数会作为表所有者运行。 (CVE-2012-0866)

- 删除SSL认证中通用名长度的任意限制 (Heikki Linnakangas)

  libpq和服务器都缩短了从SSL认证中提取的通用名为32字节。 通常这没什么坏处，除了一个意外的认证失败，但是有一些相当那以置信的情况下， 它可能允许一个证书持有人冒充另一个。受害者必须有一个正好是32字节长度的通用名， 攻击者必须说服一个信任的CA发出一份证明书，证明该通用名有那个字符串作为前缀。 冒充一个服务器也需要一些额外的开发重定向客户端连接。

- 转换写入pg_dump注释中的名字中的换行为空格 (Robert Haas)

  pg_dump对于在它的输出脚本中的SQL注释中发出的汉化的对象名是不谨慎的。 一个包含换行的名字将至少使得脚本在语句构成上不正确。 恶意的对象名将在脚本重载时引入SQL注入风险。 (CVE-2012-0868)

- 修复插入的同时做清理时的btree索引损坏 (Tom Lane)

  一个插入导致的索引页分裂有时会导致并发的运行`VACUUM`， 导致丢失它应该删除的索引条目。在删除相应的表行之后，悬空的索引项会导致错误 （如"could not read block N in file ..."）或者更糟， 在不相关的行重新插入到表中行释放的位置时得到错误的查询结

果。 这个bug自从版本8.2就已经存在了，但是因为很少重现，所以直到现在才诊断出来。 如果你怀疑在你的数据库中发生了这种情况，那么在受影响的索引上重建索引就可修复问题。

- 修复WAL重放期间共享缓冲区的瞬态归零 (Tom Lane)

  重放逻辑会有时清空然后回填共享缓冲区，这样内容会在瞬间无效。 在热备模式，这会导致并行执行的查询看到垃圾数据。 从此会出现各种症状，但是最常见的一个是"invalid memory alloc request size"。

- 修复postmaster在热备崩溃后尝试重启 (Tom Lane)

  如果在热备模式操作时任意后端进程崩溃，那么一个逻辑错误导致postmaster中止，而不是尝试重启集群。

- 修复 `CLUSTER` / `VACUUM FULL` 挂起最近更新的行拥有的toast值 (Tom Lane)

  这个监督可能会导致在这些命令期间，在toast表的索引上报告 "duplicate key value violates unique constraint"错误。

- 在改变表的所有者时，更新每行的权限，而不只是更新每表的权限 (Tom Lane)

  未能做到这一点意味着任意先前授权的行权限仍然显示为是通过旧的所有者授权的。 这意味着不管是新的所有者还是超级用户都不能撤销该现在无法追踪表的所有者的权限。

- 在 `REASSIGN OWNED` 中支持外部数据封装和外部服务器器 (Alvaro Herrera)

  如果需要改变任意这样的对象的归属，那么这个命令会带有"unexpected classid"错误失败。

- 允许 `ALTER USER/DATABASE SET` 中的一些设置是不存在的值 (Heikki Linnakangas)

  允许 `default_text_search_config` , `default_tablespace` , 和 `temp_tablespaces` 设置为不知道名字。这是因为他们可能在另外一个使用这个设置的数据库中知道，或者对于表空间来说， 因为表空间可能还未创建。相同的问题早已被 `search_path` 认识到，这些设置并不像那个一样动作。

- 避免在提交之后删除表文件有问题时的崩溃 (Tom Lane)

  删除表会导致在事务提交之后删除底层磁盘文件。如果失败（例如，因为错误的文件权限）， 那么该代码应该只是发出一个警告信息然后继续，因为退出该事务已经太晚了。 这个逻辑在版本8.4被打破，导致这样的情况引起一个PANIC和一个不可重新启动的数据库。

- 在 `DROP TABLESPACE` 的WAL重放期间从错误事件中恢复 (Tom Lane)

重放将尝试删除表空间的目录，但是有各种会失败的原因（例如，在这些目录上的不正确的所有权或权限）。 以前，重放代码会引起恐慌，致使数据库不能重新启动除非手动介入。 似乎最好是记录该问题并记录，因为删除目录失败的唯一后果是浪费一些磁盘空间。

- 修复为热备记录AccessExclusiveLocks中的竞态条件 (Simon Riggs)

  有时一个锁被记录为就像是被"transaction zero"持有一样。 这至少已知为在从属服务器上产生声明失败，或者可能造成更严重的问题。

- 在WAL重放期间甚至打包时正确的跟踪OID计数器 (Tom Lane)

  以前，OID计数器保持停留在一个高值直到系统退出重放模式。实际的结果通常为零，但是在备用服务器中，被提升到master的情况下，一旦值是必要的， 可能需要很长的一段时间促进OID计数器为一个合理的值。

- 在故障修复的开始阻止发出误导的"consistent recovery state reached"日志信息 (Heikki Linnakangas)

- 修复`pg_stat_replication`.`replay_location`的初始值 (Fujii Masao)

  之前，显示的值是错误的，直到至少一个WAL记录被重放。

- 修复附加`*`的正则表达式的逆向引用 (Tom Lane)

  不是强制一个准确的字符串匹配，而是该代码实际上接受任意满足逆向引用符号引用的模式子表达式的字符串。

  一个类似的问题仍然困扰着嵌入到一个较大的量化的表达式中的逆向引用，而不是量词的直接主体。 这将在将来的PostgreSQL版本中处理。

- 修复处理`inet`/`cidr`值的过程中最近引入的内存泄露 (Heikki Linnakangas)

  PostgreSQL的2011.12版本中的一个补丁导致在这些操作中的内存泄露， 在有些情况下可能是重要的，如在这样的字段上建立一个btree索引。

- 修复在一个SQL语言功能中`CREATE TABLE AS`/`SELECT INTO`之后的悬挂指针 (Tom Lane)

  在大多数情况下，这只会在启用声明的建立中导致声明失败，但是有可能有更糟糕的后果。

- 避免在Windows上的syslogger中两次关闭文件句柄 (MauMau)

  通常这种错误是不可见的，但是当运行在Windows的debug版本上时它会导致一个意外。

- 修复plpgsql中I/O转换相关的内存泄露 (Andres Freund, Jan Urbanski, Tom Lane)

  某些操作会泄露内存，直到当前函数的结束。

- 改善pg_dump的继承表字段的处理 (Tom Lane)

  当子字段有一个与父字段不同的缺省表达式时，pg_dump会错误的处理这种情况。 如果缺省和父字段的缺省文本上相同，但是实际上不同（例如，因为模式搜索路径不同），将不会认为是不同，所以在转储和恢复之后，允许子字段继承父字段的缺省。 当它们的父字段不是也可以恢复巧妙的错误时，子字段为 `NOT NULL` 。

- 为INSERT-style表数据修复pg_restore的direct-to-database模式 (Tom Lane)

  当使用发布日期为2011年12月或9月的pg_restore时， Direct-to-database从带有 `--inserts` 或 `--column-inserts` 选项制作的归档文件中恢复会失败，因为监督在另外一个问题的修复中。 该归档文件本身没有问题，文本模式输出也是可以的。

- 允许pg_upgrade处理包含 `regclass` 字段的表 (Bruce Momjian)

  因为pg_upgrade现在注意保存 `pg_class` OID，不再有任何原因限制。

- 当查找一个SSL客户端证书文件时使libpq忽略 `ENOTDIR` 错误 (Magnus Hagander)

  这允许建立SSL连接，尽管不带有证书，甚至用户的根目录设置为类似 `/dev/null` 的东西。

- 修复ecpg的SQLDA区域内一些更多的字段对齐问题 (Zoltan Boszormenyi)

- 在ecpg `DEALLOCATE` 语句中允许 `AT` 选项 (Michael Meskes)

  支持这个的基础构造已经有一段时间了，但是通过一个监督，仍然有一个错误检查拒绝该情况。

- 在ecpg中定义一个varchar结构时不要使用变量名 (Michael Meskes)

- 修复 `contrib/auto_explain` 的JSON输出模式以产生有效的JSON (Andrew Dunstan)

  输出在顶级使用方括号，在它原本应该使用花括号的地方。

- 修复 `contrib/intarray` 的 `int[] & int[]` 操作符中的错误 (Guillaume Lelarge)

  如果两个输入数组有相同的最小整数为1，并且两个数组中都有更小的值， 那么1将会不正确的从结果中漏掉。

- 修复 `contrib/pgcrypto` 的 `encrypt_iv()` 和 `decrypt_iv()` 中的错误检测 (Marko Kreen)

  这些函数未能报告无效输入错误的准确类型，并且对于不正确的输入会返回随机的垃圾值。

- 修复 `contrib/test_parser` 中的一字节缓冲区溢出 (Paul Guyot)

该代码将会试图比它应该读取的多读一个字节，这会导致极端情况下的崩溃。 因为 `contrib/test_parser` 只是示例代码，这对它本身来说不是一个安全问题， 但是不好的示例代码仍是坏的。

- 如果可用，在ARM上为自旋锁使用 `__sync_lock_test_and_set()` (Martin Pitt)

  这个函数替代我们以前 `SWPB` 指令的使用，该指令已经弃用了并且在ARMv6和以后的版本中不能使用了。 报告显示，老旧代码不会在最近的ARM模块明显的失败，但是简单的不互锁并发访问， 导致在多进程操作中奇怪的失败。

- 当使用接受 `-fexcess-precision=standard` 的gcc版本建立时使用该选项 (Andrew Dunstan)

  这防止了混合情形下gcc的最近版本将产生创新结果。

- 允许在FreeBSD上使用线程的Python (Chris Rees)

  以前我们的配置脚本相信这种组合不会运行；但是FreeBSD修复了这个问题，所以删除那个错误检查。

E.28. 版本 9.0.6

> 发布日期: 2011-12-05

这个版本包含各种自9.0.5以来的修复。想要获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.28.1. 迁移到版本 9.0.6

运行9.0.X的系统不需要转储/恢复。

然而，在 `information_schema.referential_constraints` 视图的定义中发现了一个长期存在的错误。如果你依赖于该视图的正确结果， 那么你应该如下第一条变更日志条目解释的那样替换它的定义。

另外，如果你是从一个早于9.0.4的版本升级而来，那么请参阅9.0.4的版本声明。

E.28.2. 修改列表

- 修复了 `information_schema.referential_constraints` 视图中的bug (Tom Lane)

  这个视图关于匹配外键约束和依赖的主键或唯一键约束不够仔细。 这会导致无法显示外键约束，或显示多次外键约束， 或宣称它依赖于与它实际依赖的约束不同的约束。

  由于该视图的定义是通过initdb安装的，只是升级将不能修复这个问题。 如果你需要在一个现有安装上修复，你可以（作为超级用户）删除 `information_schema` 模式然后通过寻源 `_SHAREDIR_` /information*schema.sql*重新创建它。 （如果不确定 `_*SHAREDIR* `在哪，运行` pg_config --sharedir`。） 必须在每个要修复的数据库充重复这个操作。

- 修复加入标量返回函数的输出的 `UPDATE` 或 `DELETE` 期间可能的损坏 (Tom Lane)

  只有在同时更新同一个目标行时会发生损坏，所以这个问题只是间歇性的出现。

- 修复WAL记录为GIN索引更新不正确的重放 (Tom Lane)

  这会导致崩溃后或在一个热备服务器上有一个瞬间不能找到索引条目。 不过，该问题可以通过对索引的下一个 `VACUUM` 修复。

- 修复 `CREATE TABLE dest AS SELECT * FROM src` 或 `INSERT INTO dest SELECT * FROM src` 期间TOAST相关的数据损坏 (Tom Lane)

如果一个表已经通过 `ALTER TABLE ADD COLUMN` 修改， 那么逐字的拷贝它的数据到另一个表在某些极端情况下会产生损坏的结果。 该问题只会在8.4和以后的版本中的确定形式下出现， 但是我们也在更早的版本中打了补丁，以防有其他代码路径会触发相同的bug。

- 修复热备启动期间可能的失败 (Simon Riggs)
- 当最初的快照不完整时更快的启动热备份 (Simon Riggs)
- 修复toast表访问陈旧的syscache条目期间的竞态条件 (Tom Lane)

  典型症状是类似"missing chunk number 0 for toast value NNNNN in pg_toast_2619" 这样的瞬态错误，这里被引用的toast表总是属于系统目录。

- 追踪用于参数缺省表达式中的条目上的函数的依赖性 (Tom Lane)

  以前，可以在没有删除或修改函数之前删除引用对象，导致使用该函数时的错误行为。 请注意，仅仅安装这个更新将不能修复丢失的依赖条目；要想修复，你需要在每一个这样的函数之后 `CREATE OR REPLACE` 。如果你有缺省依赖于非内建对象的函数，建议这样做。

- 允许设置-返回SQL函数内联多个OUT参数 (Tom Lane)
- 对于加入删除来说不要依赖延迟的唯一索引 (Tom Lane and Marti Raudsepp)

  延迟的唯一约束可能不支持内部事务，所以假定它会给出不正确的查询结果。

- 使 `DatumGetInetP()` 取出有1字节开头的inet资料，并且添加一个新的宏 `DatumGetInetPP()` (Heikki Linnakangas)

  这个改变不会影响核心代码，但是可以阻止期望 `DatumGetInetP()` 按照通常的惯例产生非压缩的数据的扩展代码中的崩溃。

- 改善 `money` 类型的输入和输出的语言环境支持 (Tom Lane)

  除了不支持所有的标准 `lc_monetary` 格式化选项之外，输入和输出函数也是不一致的，这意味着转储的 `money` 值中的语言环境不可以被重新读取。

- 不让 `transform_null_equals` 影响 `CASE foo WHEN NULL ...` 构造 (Heikki Linnakangas)

  `transform_null_equals` 只应该影响直接由用户编写的 `foo = NULL` 表达式， 不是这种形式的 `CASE` 内部产生的平等检查。

- 更改外键触发创建以便更好的支持自我参考的外键 (Tom Lane)

  对于参考它自己的表的级联外键，一个行更新将作为一个事件触发 `ON UPDATE` 触发器和 `CHECK` 触发器。必须先执行 `ON UPDATE` 触发器，否则 `CHECK` 将检查行的一个非最终状态，并且可能抛出一个不恰当的错误。不过，触发这些触发器的顺序则取决于它们的名字， 它们通常以创建的顺序排序，因为触发器自动生成的名字遵循惯

例"RI_ConstraintTrigger_NNNN"。 适当的修复需要修改该惯例，我们将在9.2中做这块，不过在现存的版本中改变它看起来是有风险的。 所以这个修复只是改变触发器的创建顺序。用户遇到这个类型的错误时应该删除并重建外键约束， 以使它的触发器获得正确的顺序。

- 当跟踪缓冲区分配率时避免浮点型下溢 (Greg Matthews)

  当它本身无害时，在特定平台上这将导致恼人的内核日志信息。

- 当在Windows下启动子进程时，保护配置文件名和行号值 (Tom Lane)

  以前，这些在 `pg_settings` 视图中显示的不正确。

- 修复ecpg的SQLDA区域中不正确的字段排列 (Zoltan Boszormenyi)

- 保持空行在psql的命令历史的命令里面 (Robert Haas)

  前者的行为可能导致问题，例如，如果一个空行从一个字符串中删除。

- 修复pg_dump以转储用户定义和自动生成类型之间的转换，比如表的行类型 (Tom Lane)

- pg_upgrade的各种修复 (Bruce Momjian)

  正确的处理排他约束，避免在Windows上的失败，不要抱怨8.4数据库中错误匹配的toast表名。

- 使用xsubpp的首选版本建立PL/Perl，不一定要操作系统的主要副本 (David Wheeler and Alex Hunsaker)

- 修复 `contrib/dict_int` 和 `contrib/dict_xsyn` 中不正确的编码 (Tom Lane)

  某些函数不正确的假设由 `palloc()` 返回的内存保证调到零位。

- 修复 `contrib/unaccent` 的配置文件解析中的各种错误 (Tom Lane)

- 查询取消 `pgstatindex()` 中的立即中断 (Robert Haas)

- 修复Mac OS X启动脚本中日志文件名的错误引用 (Sidar Lopez)

- 确保VPATH建立适当的安装所有的服务器头文件 (Peter Eisentraut)

- 缩短在冗长的错误信息中报告的文件名 (Peter Eisentraut)

  正规的建立总是只是报告包含错误信息调用的C文件的名字，但是VPATH建立以前报告一个绝对路径名。

- 修复Windows时区名称中美洲（Central America）的说明 (Tom Lane)

  映射"Central America Standard Time"到 `CST6` ，而不是 `CST6CDT` ， 因为DST在中美洲通常不是随处可见的。

- 更新时区数据文件为tzdata版本2011n，因为DST规律在 Brazil, Cuba, Fiji, Palestine, Russia, 和 Samoa改变了； 还有Alaska 和 British East Africa的历史的修正。

E.29. 版本 9.0.5

> 发布日期: 2011-09-26

这个版本包含各种自9.0.4以来的修复。要想获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.29.1. 迁移到版本 9.0.5

运行9.0.X的系统不需要转储/恢复。

另外，如果你是从一个早于9.0.6的版本升级而来，请参阅9.0.6的发布说明。

E.29.2. 修改列表

- 修复系统目录上 `VACUUM FULL` 或 `CLUSTER` 之后目录缓存失效 (Tom Lane)

  在某些情况下，重定位系统目录行到另一个位置将不会被并发服务器进程识别， 如果他们然后尝试更新那行，那么允许目录损坏的发生。最坏的情况可能是完全丢失一个表。

- 修复sinval复位处理期间不正确的操作顺序，并且保证TOAST OID保存到系统目录中 (Tom Lane)

  这些错误可能导致在一个系统目录上 `VACUUM FULL` 或 `CLUSTER` 之后的瞬时失效。

- 修复索引in-doubt HOT-updated元组的bug (Tom Lane)

  这些bug可能导致在对系统目录重建索引之后的索引损坏。不认为会影响用户的索引。

- 修复GiST索引页分裂进程中的多个bug (Heikki Linnakangas)

  发生的概率很低，但是可能会导致索引损坏。

- 修复 `tsvector_concat()` 中可能的缓冲区溢出 (Tom Lane)

  该函数可能低估了它的结果所需内存的数量，导致服务器崩溃。

- 修复处理"standalone"参数时 `xml_recv` 中的崩溃 (Tom Lane)

- 使 `pg_options_to_table` 为不带值的选项返回 NULL (Tom Lane)

  以前这样的情况会导致服务器崩溃。

- 避免在 `ANALYZE` 和SJIS-2004编码转换中可能的访问超出内存结尾 (Noah Misch)

这修复了一些非常低可能性的服务器崩溃情况。

- 保护 `pg_stat_reset_shared()` 不输入NULL (Magnus Hagander)
- 修复子事务中检测到一个恢复冲突死锁时可能的失败 (Tom Lane)
- 避免热备期间回收btree索引页时伪造的冲突 (Noah Misch, Simon Riggs)
- 如果WAL接收器仍然运行在恢复的结尾，那么关闭它 (Heikki Linnakangas)

  postmaster以前在这种情况下会恐慌，但它实际上是一个合法的情况。

- 修复relcache初始化文件无效中的竞态条件 (Tom Lane)

  有一个窗口，一个新的后端进程可能会读取旧的初始化文件， 而忽视将要告诉它该数据是陈旧的的inval信息。结果将会是目录访问中的奇异的失败， 典型的是稍后启动时"could not read block 0 in file ..."。

- 修复在GiST索引扫描结尾的内存泄露 (Tom Lane)

  执行许多独立GiST索引扫描的命令，例如在一个早已包含许多行的表上验证一个新的基于GiST的排除约束， 可能因为这个漏洞在瞬间需要大量的内存。

- 修复在进入命令字符串和 `LISTEN` 是活跃的时必须做编码转换时的内存泄露 (Tom Lane)
- 修复支持可持有的游标和plpgsql的 `RETURN NEXT` 命令的tuplestores中的错误的内存计算（导致可能的内存膨胀） (Tom Lane)
- 修复 `BEFORE` 和 `AFTER` 触发器都存在时的触发器 `WHEN` 条件 (Tom Lane)

  如果已经有一个 `BEFORE ROW` 触发器为同一个更新触发了，那么为 `AFTER ROW UPDATE` 评估 `WHEN` 条件可能会崩溃。

- 修复构建一个大的、有损耗的位图时的性能问题 (Tom Lane)
- 修复为唯一字段的连接选择性估计 (Tom Lane)

  这修复了一个可能导致欠佳的估计连接结果大小的错误的启发式规划器。

- 修复只在子查询目标列表中出现的嵌套的PlaceHolderVar表达式 (Tom Lane)

  这个错误可能导致外连接的输出不正确的显示为NULL。

- 允许规划器认为空父表真的是空的 (Tom Lane)

  通常一个空表为了规划目的会假设有一个特定的最小尺寸； 但是这个启发对于通常永久为空的继承体系的父表来说看起来弊大于利。

- 允许嵌套的 `EXISTS` 查询适当的最优化 (Tom Lane)
- 修复array-和path-creating函数以确保填充的字节为0 (Tom Lane)

这避免了规划器认为语义上相等的常量不等，导致欠佳的最优化的情况。

- 修复 `EXPLAIN` 以处理内部索引扫描辅助方案内的控制结果节点 (Tom Lane)

  这种监督通常的症状是"bogus varno"错误。

- 修复 `_indexedcol_ IS NULL` 条件的btree预处理 (Dean Rasheed)

  如果与任意其他类型的btree可索引条件在同一个索引字段上结合，那么这样一个条件可能是不满足的。 该情况在9.0.0和之后的版本中处理的不正确，导致查询输出到本该没有的地方。

- 绕开破坏WAL重放的gcc 4.6.0 bug (Tom Lane)

  这会导致在服务器崩溃后丢失已提交的事务。

- 修复视图中的 `VALUES` 的转储bug (Tom Lane)
- 禁止序列上的 `SELECT FOR UPDATE/SHARE` (Tom Lane)

  这个操作不会像预期的那样工作，并且会导致失败。

- 修复 `VACUUM` 以便它总是更新 `pg_class`.`reltuples`/`relpages` (Tom Lane)

  这修复了一些自动清理可能使得越来越多的什么时候清理表的决策欠佳的情形。

- 当计算一个哈希表的大小时防止整数溢出 (Tom Lan)
- 修复 `CLUSTER` 试图访问早已删除的TOAST数据的情况 (Tom Lane)
- 修复初始验证事务期间过早的超时失败 (Tom Lane)
- 修复为"peer"认证使用证书控制信息中的可移植性错误 (Tom Lane)
- 修复需要多次往返时的SSPI登陆 (Ahmed Shinwari, Magnus Hagander)

  这个问题的典型症状是在SSPI登陆期间的"The function requested is not supported"错误。

- 修复添加一个新的自定义变量类的变量到 `postgresql.conf` 时的失败 (Tom Lane)
- 如果 `pg_hba.conf` 包含 `hostssl` 但是SSL是禁用的时，抛出一个错误 (Tom Lane)

  这被认为是更加用户友好的行为，以前这样的行都是默默地忽视。

- 修复 `DROP OWNED BY` 试图删除序列上的缺省权限时的失败 (Shigeru Hanada)
- 修复 `pg_srand48` 种子初始化中的打字错误 (Andres Freund)

这导致未能使用所有提供的种子。这个函数没有用在大多数的平台上（只是那些没有 `srandom` 的）， 并且一个比预期较少随机的种子看起来在任何情况下潜在的安全风险都很小。

- 避免 `LIMIT` 和 `OFFSET` 值的和超过2^63时的整数溢出 (Heikki Linnakangas)
- 添加溢出检查到 `generate_series()` 的 `int4` 和 `int8` 版本 (Robert Haas)
- 修复在 `to_char()` 中消除尾随的零 (Marti Raudsepp)

  在 `FM` 格式中并且在小数点后没有数字位，小数点左边的零可能会被不正确的消除。

- 修复 `pg_size_pretty()` 以避免输入接近2^63而溢出 (Tom Lane)
- 减少记录值中typmod匹配的plpgsql的检查 (Tom Lane)

  一个过度的检查可能会导致丢弃应该保持的长度修饰符。

- 正确的处理initdb期间本地名中的引号 (Heikki Linnakangas)

  该情况可能出现一些Windows地区，如"People's Republic of China"。

- 在pg_upgrade中，避免转储孤立的临时表 (Bruce Momjian)

  这防止了表OID分配在旧的和新的安装中不同步的情况。

- 修复pg_upgrade以在从8.3升级而来期间保持toast表的relfrozenxids (Bruce Momjian)

  未能做到这点会导致 `pg_clog` 文件在升级之后很快被删掉。

- 在pg_upgrade中，修复 `-l` (log)选项以在Windows上运行 (Bruce Momjian)
- 在pg_ctl中，在Windows上支持服务注册的静音模式 (MauMau)
- 修复从一个不同的文件 `COPY` 期间psql的脚本文件行号计数 (Tom Lane)
- 为 `standard_conforming_strings` 修复pg_restore 的direct-to-database模式 (Tom Lane)

  pg_restore在从一个由 `standard_conforming_strings` 设置为 `on` 制作的归档文件中直接恢复到数据库服务器时可能会发出不正确的命令。

- 对于并行pg_restore的不支持的情况更加用户友好 (Tom Lane)

  这个改变确保了这样的情况能被检测到，并且在采取恢复动作之前被报告。

- 修复libpq的LDAP服务查找代码中的write-past-buffer-end和内存泄露 (Albe Laurenz)
- 在libpq中，避免使用无闭塞的I/O和SSL连接时的失败 (Martin Pihlak, Tom Lane)
- 改善libpq对连接启动时失败的处理 (Tom Lane)

  特别的，现在对服务器在SSL连接启动时失败的 `fork()` 报告的响应更加理智。

- 改善libpq对于SSL失败的错误报告 (Tom Lane)
- 修复`PQsetvalue()`以避免添加一个新的元组到一个最初从服务器查询获得的`PGresult`时可能的崩溃 (Andrew Chernow)
- 使ecpglib写`double`值带有15位数字精度 (Akira Kurosawa)
- 在ecpglib中，确保`LC_NUMERIC`设置在错误之后恢复 (Michael Meskes)
- 为blowfish有符号字符的错误应用逆向修复 (CVE-2011-2483)

  `contrib/pg_crypto`的blowfish加密代码在char是有符号的平台（大多数是） 上可能给出错误的结果，导致加密的口令比原有的强度要弱。
- 修复`contrib/seg`中的内存泄露 (Heikki Linnakangas)
- 修复`pgstatindex()`以对于空索引给出一致的结果 (Tom Lane)
- 允许使用perl 5.14建立 (Alex Hunsaker)
- 修复建立和安装的文件路径包含空格时的各种问题 (Tom Lane)
- 更新时区数据文件到tzdata 版本 2011i，因为DST法律在以下地区发生改变： Canada, Egypt, Russia, Samoa, 和 South Sudan。

E.30. 版本 9.0.4

> 发布日期: 2011-04-18

这个版本包含各种自9.0.3以来的修复。要想获取9.0主版本的新特性信息， 请参阅Section E.34。

E.30.1. 迁移到版本 9.0.4

运行9.0.X版本的用户不需要转储/恢复。

不过，如果您的安装是通过运行pg_upgrade从一个先前的主版本升级而来， 您应该采取措施防止由于pg_upgrade中一个现在修复的bug引起可能的数据丢失。 建议的解决方案是在所有的TOAST表上运行 `VACUUM FREEZE` 。 更多信息可以参考 http://wiki.postgresql.org/wiki/20110408pg_upgrade_fix。

E.30.2. 修改列表

- 修复pg_upgrade对TOAST表的处理 (Bruce Momjian)

  在pg_upgrade期间，TOAST表的 `pg_class` . `relfrozenxid` 值没有正确的拷贝到新的安装中。这可能稍后导致他们仍然需要在TOAST表中验证元组时， `pg_clog` 文件被丢弃，导致"could not access status of transaction"失败。

  这个错误造成了用pg_upgrade升级的安装的数据丢失的重大风险。 这个补丁纠正了将来使用pg_upgrade的问题， 但是没有本质上解决已经用错误版本的pg_upgrade处理过的安装中的问题。

- 废止错误的"PD_ALL_VISIBLE flag was incorrectly set"警告 (Heikki Linnakangas)

  `VACUUM` 有时会发出这个警告，实际上是有效的。

- 为热备份冲突情况使用更好的SQLSTATE错误代码 (Tatsuo Ishii and Simon Riggs)

  所有可重试的冲突错误现在有一段错误代码表明重试是可能的。另外， 由于数据库在master上删除引起的会话关闭现在报告为 `ERRCODE_DATABASE_DROPPED` ， 而不是 `ERRCODE_ADMIN_SHUTDOWN` ，所以连接池能正确的处理该情形。

- 防止间歇性挂在启动进程和后端写进程的交互中 (Simon Riggs)

  这影响在非热备份情况下的恢复。

- 不允许包括复合类型本身 (Tom Lane)

这阻止了在服务器中处理复合类型时会无限递归的情形。当可能有一些这样的机构的使用时， 它们似乎不能足够令人信服的证明所需要的努力，以确保它总是安全的工作。

- 在目录缓存初始化期间避免潜在的死锁 (Nikhil Sontakke)

  在某些情况下，缓存加载代码在锁住索引的目录之前会在系统索引上请求共享锁。 这可能会死锁对进程试图请求其他独占锁，更标准的顺序。

- 修复当有并发更新到目标元组时 `BEFORE ROW UPDATE` 触发器处理中的悬空指针问题 (Tom Lane)

  观察到这个错误在试图做 `UPDATE RETURNING ctid` 时， 会导致间歇性的 "cannot extract system attribute from virtual tuple"失败。 有一个非常小的概率会发生更严重的错误，如为更新的元组生成不正确的索引条目。

- 当等待一个表的延迟触发的事件时，不允许 `DROP TABLE` (Tom Lane)

  以前的 `DROP` 会通过，导致触发器最终触发时的 "could not open relation with OID nnn"错误。

- 在 `pg_hba.conf` 中允许"replication"作为一个用户名 (Andrew Dunstan)

  "replication"在数据库名字字段中是特殊的，但是它在用户名字段中被错误的认为也是特殊的。

- 阻止GEQO最优化期间由WHERE条件常量错误触发的崩溃 (Tom Lane)

- 改善规划者处理半连接和反连接的情况 (Tom Lane)

- 修复子SELECT中 `SELECT FOR UPDATE` 的处理 (Tom Lane)

  这个错误通常导致"cannot extract system attribute from virtual tuple"错误。

- 修复文本搜索的选择性估计以计算NULL (Jesper Krogh)

- 修复get_actual_variable_range()以支持通过索引指导插件假设的索引注入 (Gurjeet Singh)

- 修复包含数组分片的PL/Python内存泄露 (Daniel Popowich)

- 当用户的根目录不可用时允许libpq的SSL成功初始化 (Tom Lane)

  如果是SSL模式，那么根证书文件不是必须的，也就不需要失败。 这个改变恢复到9.0之前版本的行为。

- 修复libpq为在 `conninfo_array_parse` 中检测到的错误返回有用的错误信息 (Joseph Adams)

  打字错误导致库返回NULL给应用，而不是包含错误信息的 `PGconn` 构成。

- 修复ecpg预处理器对浮点数常量的处理 (Heikki Linnakangas)
- 修复并发的pg_restore以正确的处理POST_DATA条目上的注释 (Arnd Hannemann)
- 修复pg_restore以处理TOC文件中较长的行（超过 1KB） (Tom Lane)
- 针对由过度热情的编译器优化和被零除引起的崩溃投入更多的保障 (Aurelien Jarno)
- 支持在MIPS上的FreeBSD和OpenBSD中使用dlopen() (Tom Lane)

  有一个硬链接假设这个系统功能在这些系统的MIPS硬件上不可用。 那么使用一个编译时测试，因为最近的版本有这个功能。

- 修复HP-UX上的编译错误 (Heikki Linnakangas)
- 避免在启动过程中非常早的尝试写入Windows控制台时的崩溃 (Rushabh Lathia)
- 支持为Windows用MinGW 64位编译器建立 (Andrew Dunstan)
- 修复Windows上libintl的版本不兼容问题 (Hiroshi Inoue)
- 修复xcopy在Windows生成脚本中的使用，以在Windows 7下正确的运转 (Andrew Dunstan)

  这只影响生成脚本，不影响安装或使用。

- 修复在Cygwin上被pg_regress使用的路径分隔符 (Andrew Dunstan)
- 更新时区数据文件到tzdata版本2011f，因为DST规律在以下地区发生了改变： Chile, Cuba, Falkland Islands, Morocco, Samoa, 和 Turkey；还有South Australia, Alaska, 和 Hawaii的历史修正。

E.31. 版本 9.0.3

> 发布日期: 2011-01-31

这个版本包含各种自9.0.2以来的修复。要想获得关于9.0主版本的新特性， 请参阅Section E.34。

E.31.1. 迁移到版本 9.0.3

运行9.0.X版本的用户不需要转储/恢复。

E.31.2. 修改列表

- 在退出walreceiver之前，确保所有接收到的WAL同步到磁盘 (Heikki Linnakangas)

  否则备用服务器会重放一些未同步的WAL，想象的到，如果系统正好在此时崩溃，则会导致数据损坏。

- 避免walreceiver中过度的同步活动 (Heikki Linnakangas)

- 需要时使 `ALTER TABLE` 的唯一性和排除约束重新生效 (Noah Misch)

  这在9.0中，由于试图在 `VACUUM FULL` 和 `CLUSTER` 期间抑制重新生效而坏掉了， 但是无意间也影响到了 `ALTER TABLE` 。

- 为继承树的 `UPDATE` 修复EvalPlanQual，该继承树中的表并不都是相似的 (Tom Lane)

  表行类型（包括只在一些子表中出现的少量的字段）的任何变化都将使EvalPlanQual代码混乱， 导致错误行为甚至崩溃。因为EvalPlanQual只在并发更新到相同的行时执行，该问题只能间歇的看到。

- 避免 `EXPLAIN` 试图显示一个简单形式的 `CASE` 表达式时的失败 (Tom Lane)

  如果 `CASE` 的测试表达式是一个常量，那么规划器将简化 `CASE` 为一个困惑表达式显示代码的形式， 导致"unexpected CASE WHEN clause"错误。

- 修复现有下标范围之前的数组切片的分配 (Tom Lane)

  如果新添加的下标和原先存在的下标之间存在一个缺口， 代码错误估算了需要多少条目从老数组的空位图中拷贝，可能导致数据损坏或崩溃。

- 避免为了非常远的日期值在规划器中意外的转换溢出 (Tom Lane)

`date` 类型比 `timestamp` 类型支持更大范围的日期， 但是规划器假设它可以总是不受惩罚的转换一个日期为时间戳。

- 修复一个数组包含空条目时的PL/Python损坏 (Alex Hunsaker)
- 为定义一个数组维度的常量删除ecpg的固定的长度限制 (Michael Meskes)
- 修复包含 `... & !(subexpression) | ...` 的 `tsquery` 值的错误解析 (Tom Lane)

  包含这些操作符的组合的查询没有正确的执行。相同的错误存在于 `contrib/intarray` 的 `query_int` 类型和 `contrib/ltree` 的 `ltxtquery` 类型。

- 修复 `contrib/intarray` 的 `query_int` 类型输入函数中的缓冲区溢出 (Apple)

  这个错误是一个安全风险，因为该函数的返回地址会被重写。 感谢Apple Inc的安全团队报告这个问题并提供了该修复。 (CVE-2010-4015)

- 修复 `contrib/seg` 的GiST picksplit算法中的错误 (Alexander Korotkov)

  这可能会导致大量的低效，尽管不是实际上错误的答案，在 `seg` 字段上的GiST索引中。如果你有这样的一个索引，那么考虑在安装这个更新之后 `REINDEX` 它。 （这与之前更新的 `contrib/cube` 中修复的错误相同。）

E.32. 版本 9.0.2

> 发布日期: 2010-12-16

这个版本包含各种自9.0.1以来的修复。要想获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.32.1. 迁移到版本 9.0.2

运行9.0.X版本的用户不需要转储/恢复。

E.32.2. 修改列表

- 强制缺省 `wal_sync_method` 在Linux上为 `fdatasync` (Tom Lane, Marti Raudsepp)

  linux上的缺省实际上已经 `fdatasync` 很多年了，但是最近的内核改变导致 PostgreSQL选择了 `open_datasync` 。 这个选择不会导致任何性能改进，并且在某些文件系统上导致彻底的失败， 尤其是带有 `data=journal` 接口选项的 `ext4` 。

- 修复热备重放期间的"too many KnownAssignedXids"错误 (Heikki Linnakangas)

- 修复热备期间锁捕获中的竞态条件 (Simon Riggs)

- 避免热备期间不必要的冲突 (Simon Riggs)

  这修复了一些认为重放与备用查询相冲突的情况（导致重放延迟或可能取消查询）， 但是没有真正的冲突。

- 修复为GIN索引WAL重放逻辑中的各种错误 (Tom Lane)

  这会导致复制期间"bad buffer id: 0"失败或索引内容的损坏。

- 修复起始检查点WAL记录和它的重做点不在同一个WAL段中时的从基础备份中恢复 (Jeff Davis)

- 修复在创建主数据库集群之后立即启用流复制时的极端情况错误 (Heikki Linnakangas)

- 修复多个workers持续活跃了很长一段时间时autovacuum workers的持续放缓 (Tom Lane)

  autovacuum worker如果处理了足够的表，那么它有效的 `vacuum_cost_limit` 可以下降到接近零， 导致它运行极其的慢。

- 修复autovacuum启动器中长期的内存泄露 (Alvaro Herrera)

- 避免试图从一个事务外面报告一个迫近的事务概括条件时的失败 (Tom Lane)

  这个监督防止了事务概括太近之后的恢复，因为数据库启动处理会失败。

- 添加支持检测寄存器栈在 `IA64` 上溢出 (Tom Lane)

  `IA64` 体系结构有两个硬件堆栈。全面预防堆栈溢出失败都需要检查。

- 添加一个 `copyObject()` 中堆栈溢出的检查 (Tom Lane)

  某些代码路径会由于给出足够复杂的查询堆栈溢出而崩溃。

- 修复临时GiST索引中页面分裂的检测 (Heikki Linnakangas)

  在一个临时索引中有"并发的"页分裂是可能的，例如有一个打开的游标在插入时扫描索引。 GiST未能检测这种情况，并且因此可能当游标的执行继续时交付错误的结果。

- 修复早期连接过程期间的错误检查 (Tom Lane)

  太多子进程的检查在某些情况下是跳过的， 可能会导致试图添加新的子进程到固定大小的数组时postmaster崩溃。

- 提高窗口函数的效率 (Tom Lane)

  某些情况需要提前读取大量的元组，但是 `work_mem` 足够大的允许它们都在内存中进行，这会出乎意料的慢。尤其是 `percent_rank()`，`cume_dist()` 和 `ntile()` 有这个问题。

- 避免 `ANALYZE` 复杂的索引表达式时的内存泄露 (Tom Lane)

- 确保使用整行Var的索引依赖于它的表 (Tom Lane)

  一个像 `create index i on t (foo(t.*))` 这样声明的索引， 当它的表被删除时它不会自动被删除。

- 在 `DROP OWNED BY` 中为删除属于一个用户的外部数据封装/服务器权限添加缺失支持 (Heikki Linnakangas)

- 不要用多个 `OUT` 参数"inline"一个SQL函数 (Tom Lane)

  这避免了由于丢失关于预期的结果行类型的信息而引起的可能的崩溃。

- 修复内联一个参数列表包含一个引用可内联的用户函数的设置返回函数时的崩溃 (Tom Lane)

- 如果 `ORDER BY`，`LIMIT`，`FOR UPDATE`，或 `WITH` 附属于 `INSERT ... VALUES` 的 `VALUES` 部分则行为正确 (Tom Lane)

- 使 `OFF` 关键字无限制 (Heikki Linnakangas)

这阻止了在PL/pgSQL中使用 `off` 作为变量名的问题。 在9.0之前是好用的，但是现在打破了，因为PL/pgSQL 现在将所有的内核保留字作为保留的。

- 修复 `COALESCE()` 表达式的常量部分 (Tom Lane)

  规划器有时试图计算子表达式，实际上从未达成，可能导致预想不到的错误。

- 修复"could not find pathkey item to sort"规划器带有整行Vars比较失败 (Tom Lane)

- 修复连接接受（ `accept()` 或在它之后立即做的调用中的一个）失败时的postmaster崩溃， 并且postmaster是用GSSAPI支持编译的 (Alexander Chernikov)

- 在从RADIUS认证服务器接收到一个无效的响应包之后重试 (Magnus Hagander)

  这修复了一个低风险潜在拒绝服务的情况。

- 修复了 `log_temp_files` 活跃时错过了删除临时文件 (Tom Lane)

  如果在试图发出日志消息时发生错误，删除文件没有做，导致临时文件的积累。

- 为 `InhRelation` 节点添加打印功能 (Tom Lane)

  这避免了启动 `debug_print_parse` 和执行确定类型的查询时的失败。

- 修复了从一个点到一个水平线段的距离的不正确的计算 (Tom Lane)

  这个错误影响几个不同的几何距离测量操作。

- 修复ecpg中交易状态的不正确的计算 (Itagaki Takahiro)

- 修复psql的Unicode逃逸支持中的错误 (Tom Lane)

- 当归档包含许多大对象（blobs）时加速并行的pg_restore (Tom Lane)

- 修复PL/pgSQL处理"简单的"表达式，以不在递归或错误恢复的情况下失败 (Tom Lane)

- 修复PL/pgSQL错误报告no-such-column的情况 (Tom Lane)

  截止到9.0，有时会报告"missing FROM-clause entry for table foo"， 而此时"record foo has no field bar"将会更合适。

- 修复分配到元组字段时PL/Python遵从typmod（也就是，长度或精度限制） (Tom Lane)

  这修复了从8.4的回归。

- 修复PL/Python处理设置返回函数 (Jan Urbanski)

  试图在迭代器中调用SPI函数生成一个设置结果将会失败。

- 修复 `contrib/cube` 的GiST picksplit算法中的错误 (Alexander Korotkov)

这可能会导致大量的低效，尽管不是实际上错误的答案，在 `cube` 字段上的GiST索引中。 如果你有这样的一个索引，那么考虑在安装这个更新之后 `REINDEX` 它。

- 不要在 `contrib/dblink` 中发出"identifier will be truncated"通知， 除非创建新的连接时 (Itagaki Takahiro)
- 修复 `contrib/pgcrypto` 中丢失公钥潜在的内核转储 (Marti Raudsepp)
- 修复 `contrib/pg_upgrade` 中的缓冲区溢出 (Hernan Gonzalez)
- 修复 `contrib/xml2` 的XPath查询函数中的内存泄露 (Tom Lane)
- 更新失去数据文件为tzdata版本2010o，因为DST规律在Fiji和Samoa发生了改变； 还有Hong Kong的历史修正。

E.33. 版本 9.0.1

> 发布日期: 2010-10-04

这个版本包含各种自9.0.0以来的修复。要想获得关于9.0主版本的新特性信息， 请参阅Section E.34。

E.33.1. 迁移到版本 9.0.1

运行9.0.X版本的用户不需要转储/恢复。

E.33.2. 修改列表

- 在PL/Perl和PL/Tcl中为每个调用的SQL userid使用单独的解释器 (Tom Lane)

  这个改变阻止由破坏稍后在相同的会话中不同的SQL用户身份执行的Perl或Tcl代码引起的安全问题 （例如，在 `SECURITY DEFINER` 函数中）。大多数脚本语言提供多种可能做的方式， 如重新定义被目标函数调用的标准函数或操作符。没有这个改变， 任何拥有Perl或Tcl语言使用权限的SQL用户都可以用目标函数所有者的SQL特权从本质上做任何事情。

  这个改变的成本是Prel和Tcl函数之间有意的沟通变得更加困难。为了提供一个逃逸出口， PL/PerlU和PL/TclU函数继续每个会话只使用过一个解释器。

  有可能要求提供可信任的执行的第三方程序语言有相同的安全问题。 我们建议为了安全性关键目的，联系你依赖的PL的作者。

  我们感谢Tim Bunce提出这个问题 (CVE-2010-3433)。

- 改善 `pg_get_expr()` 安全修复，以便该函数仍然可以用作子查询的输出 (Tom Lane)

- 修复占位符的位置不正确的评价 (Tom Lane)

  这个错误会导致查询输出应该为空时为非空， 如果外连接的内侧是一个在它的输出列表中带有非严格表达式的子查询。

- 修复连接除去占位符表达式的处理 (Tom Lane)

- 修复可能的 `UNION ALL` 成员关系的副本扫描 (Tom Lane)

- 阻止未监听之后ProcessIncomingNotify()中的无限循环 (Jeff Davis)

- 阻止show_session_authorization()在autovacuum进程中崩溃 (Tom Lane)

- 重新允许儒略日期的输入在0001-01-01 AD之前 (Tom Lane)

  像`'J100000'::date`这样的输入在8.4之前是可以工作的， 但是添加错误检查后无意的打破了。

- 使psql识别`DISCARD ALL`为一个命令， 该命令不应该在自动提交关闭模式下包含在事务块中 (Itagaki Takahiro)

- 更新建立基础结构和文档以体现源代码仓库从CVS搬至Git (Itagaki Takahiro)

E.34. 版本 9.0

> 发布日期: 2010-09-20

E.34.1. 概述

这个PostgreSQL版本添加了请求多年的特性，例如易于使用的复制， 大规模权限改变设施，和匿名代码块。而以前的主版本在这些范围是保守的， 这个版本显示了一个大胆的新的渴望提供的设施，新的和现有的PostgreSQL 用户将包含在内。这些都做到了，只是有少许的不兼容。主要的增强功能包括：

- 内建复制基于日志传送。这个发展包括两个特点：流复制，允许持续归档(WAL) 文件，在网络连接上流出到备用服务器；热备份，允许持续归档备用服务器以执行只读的查询。 有效效应是支持一个主服务器带有多个只读从服务器。
- 更简单的数据库对象权限管理。 `GRANT` / `REVOKE IN SCHEMA` 支持在现有的对象上的大量的权限更改，而 `ALTER DEFAULT PRIVILEGES` 允许控制将来创建的对象的权限。 大对象（BLOBs）现在也支持权限管理。
- 明显提高存储过程支持。 `DO` 语句支持ad-hoc或"anonymous"代码块。函数现在可以用命名的参数来调用。 PL/pgSQL现在是缺省安装的， PL/Perl和PL/Python 已经从几个方面增强了，包括对Python3的支持。
- 全面支持64位Windows
- 更高级的报告查询，包括附加的窗口选项(`PRECEDING` 和 `FOLLOWING`) 和控制值以什么顺序供应给聚集函数的能力。
- 新增触发特征，包括SQL标准兼容每字段触发器 和条件触发器执行。
- 可延期的唯一约束。 大量的更新到唯一键现在不带有欺骗是可能的了。
- 排除约束。 这些提供了一个唯一约束的广义版本，允许执行复杂的条件。
- 新增和加强的安全特性，包括RADIUS认证，LDAP认证改进，和一个新的贡献模块 `passwordcheck` 测试密码强度。
- 新增高性能实现 `LISTEN` / `NOTIFY` 特性。 等待发生的事件现在存储在一个基于内存的队列中，而不是一个表中。 还有，"payload"字符串可以被每个事件发送，而不是和以前一样只能是一个事件名传送。
- 新增 `VACUUM FULL` 的实现。 这个命令现在改写全部的表和索引，而不是移动个别的行到紧密空间。 它实质上在大多数情况下更快，并且不再导致索引膨胀。

- 新增贡献模块 `pg_upgrade`， 支持从8.3或8.4到9.0的就地升级。
- 对于特定类型的查询的多种性能增强，包括删除不需要的连接。 这帮助优化了一些自动生成的查询，比如由对象关系映射（ORMs）产生的那些。
- `EXPLAIN` 增强。 该输出现在可以用在JSON, XML, 或YAML格式中， 并且包括缓冲区利用和以前不可用的其他数据。
- `hstore` 增强，包括新功能和更大的数据容量。

以上的条目在下面的章节中有更详细的介绍。

E.34.2. 迁移到版本 9.0

使用pg_dump或使用pg_upgrade 的转储/恢复，对于那些想要从任何以前的版本迁移数据的人来说是必须的。

版本9.0包含若干有选择的突破向后兼容的改变，为了支持新特性和编码质量改善。 特别的，使得PL/pgSQL、Point-In-Time Recovery (PITR)、或Warm Standby广泛应用的用户， 应该测试他们的应用，因为在这些地方有轻微的用户可见的改变。 观察下面的不兼容：

E.34.2.1. 服务器设置

- 删除服务器参数 `add_missing_from`， 该参数缺省为off已经很多年了 (Tom Lane)
- 删除服务器参数 `regex_flavor`， 该参数缺省为 `advanced` 已经很多年了 (Tom Lane)
- `archive_mode` 现在只影响 `archive_command` ； 一个新的设置， `wal_level`， 影响预写式日志的内容 (Heikki Linnakangas)
- `log_temp_files` 现在使用缺省文件尺寸单位kilobytes (Robert Haas)

E.34.2.2. 查询

- 当查询一个parent table时， 不要在作为查询一部分扫描的子表上做任何单独的权限检查 (Peter Eisentraut)

  SQL标准声明这个行为，并且它在实际上比以前在每个子表上和父表一样检查权限的行为更加方便。

E.34.2.3. 数据类型

- `bytea` 输出现在缺省以十六进制格式显示 (Peter Eisentraut)

  服务器参数 `bytea_output` 可以用来选择传统的输出格式，如果为了兼容需要。

- 数组输入现在只考虑纯ASCII空白字符潜在的忽略；它将永不忽略非ASCII字符， 即使它们根据某些语言环境是空白 (Tom Lane)

  这避免了根据服务器本地设置，数组值会被不同的解释的某些极端情况。

- 改善标准兼容 `SIMILAR TO` 模式和SQL风格 `substring()` 模式 (Tom Lane)

  这包括将 `?` 和 `{...}` 处理为模式元字符， 而它们以前只是简单的文字字符；这相当于在SQL:2008中添加的新特性。 还有， `^` 和 `$` 现在视为简单的文字字符； 以前它们视为元字符，好像模式遵从POSIX而不是SQL规则。 还有，SQL标准的 `substring()` ，使用圆括号以嵌套， 不再干扰子字符串的捕获。还有，括号表达式（字符类）的处理现在更加标准兼容了。

- 根据SQL标准，在为位字符串的3参数 `substring()` 中拒绝负的长度值 (Tom Lane)

- 当减少分数秒精度时使 `date_trunc` 截断而不是圆整 (Tom Lane)

  代码对于基于整型的日期/时间总是这样动作。现在基于浮点数的日期/时间行为相似。

E.34.2.4. 对象重命名

- 当一个子表从多个无关的父表中继承相同的字段时，在 `RENAME` 期间严格执行字段名的一致性 (KaiGai Kohei)

- 当基础表字段重命名时不再自动重命名索引和索引字段 (Tom Lane)

  超级管理员仍然可以手动重命名这样的索引和字段。这个改变将需要更新JDBC驱动， 和可能的其他驱动，以便唯一索引在重命名之后正确的重组。

- `CREATE OR REPLACE FUNCTION` 可以不再改变函数参数声明的名字 (Pavel Stehule)

  为了避免在命名的参数调用中创建歧义，不再允许在一个现有函数的声明中改变输入参数的别名 （尽管名字仍然可以分配给以前未命名的参数）。要那样做，你现在必须 `DROP` 并重新创建该函数。

E.34.2.5. PL/pgSQL

- 如果变量名与查询中使用的字段名冲突，PL/pgSQL现在抛出一个错误 (Tom Lane)

  以前的行为是绑定歧义名为PL/pgSQL变量优先于查询字段，这经常导致令人吃惊的错误行为。 抛出一个错误允许简单的检测引起歧义的情景。虽然建议函数遇到这种类型的错误时修改为移除该冲突， 但是如果需要，旧的行为可以被修复，通过配置参数 `plpgsql.variable_conflict` ， 或通过每个函数的选项 `#variable_conflict` 。

- PL/pgSQL不再允许匹配特定SQL保留字的变量名 (Tom Lane)

这是校正PL/pgSQL分析器更密切的匹配内核SQL分析器的一个后果。如果需要， 变量名可以用双引号引起来以避免这种限制。

- PL/pgSQL现在要求复合结果的字段和基础类型一样匹配预期的类型修饰符 (Pavel Stehule, Tom Lane)

  例如，如果结果类型的字段声明为 `NUMERIC(30,2)` ，那么它不再适用于在该字段中返回一个 其他精度的 `NUMERIC` 。以前的版本忽略了检查类型修饰符， 并且因此允许结果行实际上并不符合声明的限制。

- PL/pgSQL现在对待选择到复合字段更加一致 (Tom Lane)

  以前，像 `SELECT ... INTO` `_rec_` . `_fld_` FROM ... 这样的声明被当做是标量赋值，即使记录字段 `_fld_` 是复合类型。 现在，它被当做记录赋值，当 `INTO` 目标是一个复合类型的规则变量时也是一样的。 所以被赋予到字段的子字段的值应该被写为 `SELECT` 列表的单独的列， 不是在以前版本中 `ROW(...)` 构造那样。

  如果你需要一个在9.0和以前版本中都能运行的方式来做这点，你可以像这样来写： `_rec_` . `_fld_` := ROW(...) FROM ...。

- 删除PL/pgSQL的 `RENAME` 声明 (Tom Lane)

  取代 `RENAME` ， 使用 `ALIAS` ， 现在可以为任意变量创建一个别名，不只是和以前一样的美元符号参数名称(如 `$1`)。

E.34.2.6. 其他的不兼容

- 不赞成使用 => 作为一个操作符名称 (Robert Haas)

  将来版本的PostgreSQL可能完全拒绝这种操作符名， 为了支持命名函数参数的SQL标准符号。目前，仍然是允许的， 但是定义这样的一个操作符时会发出警告。

- 删除对没有64位整数数据类型工作的平台的支持 (Tom Lane)

  认为所有仍然支持的平台有64位整数数据类型工作。

E.34.3. 修改列表

版本9.0有一个前所未有的新主版本特性的数量，超过200个增强、 改善、新增命令、新增函数和其他修改。

E.34.3.1. 服务器

E.34.3.1.1. 连续存档和流复制

PostgreSQL的现有备用服务器性能已经扩展为支持在备用服务器上只读查询和大大减少主从服务器之间的延迟。 对于很多用户，这将是流复制的一个有用和低管理的形式，要么为了高可用性，要么为了水平伸缩性。

- 允许备用服务器接受只读查询 (Simon Riggs, Heikki Linnakangas)

  这个特性称为热备份。有新的 `postgresql.conf` 和 `recovery.conf` 设置控制这个特性，还有大量的文档

- 允许预写式日志(WAL)数据流到备用服务器 (Fujii Masao, Heikki Linnakangas)

  这个特性称为流复制。以前的WAL数据只能以整个WAL 文件（通常每个16百万字节）为基本单位被送到备用服务器。 流复制消除低效，并且允许在主服务器上的更新以非常小的延迟传送到备用服务器上。 有新的 `postgresql.conf` 和 `recovery.conf` 设置控制这个特性， 还有大量的文档。

- 添加 `pg_last_xlog_receive_location()` 和 `pg_last_xlog_replay_location()`， 可以用来监控备用服务器的 WAL活动 (Simon Riggs, Fujii Masao, Heikki Linnakangas)

E.34.3.1.2. 性能

- 允许通过 `ALTER TABLESPACE ... SET/RESET` 为连续的和随机的页面成本估算 (`seq_page_cost` / `random_page_cost`) 设置每个表空间的值 (Robert Haas)

- 提高性能和连接查询中的EvalPlanQual再核查 (Tom Lane)

  包含连接的 `UPDATE` , `DELETE` , 和 `SELECT FOR UPDATE/SHARE` 查询现在在遇到最近更新的行时表现的更好。

- 当表早些时候在同一个事务中被创建或截断时，改善 `TRUNCATE` 的性能 (Tom Lane)

- 提高找到继承子表的性能 (Tom Lane)

E.34.3.1.3. 优化

- 删除不需要的外连接 (Robert Haas)

  内侧的外连接是唯一的，并且不被连接的上级引用，因此现在删除了。 这将加速许多自动生成的查询，如那些通过对象关系映射(ORMs)创建的查询。

- 允许 `IS NOT NULL` 限制使用索引 (Tom Lane)

  这对于在包含许多空值的索引中发现 `MAX()` / `MIN()` 值是尤其有用的。

- 改善优化器为 `DISTINCT` 选择何时使用物化节点，何时使用排序和散列 (Tom Lane)

- 改善优化器为包含 `boolean` `<>` 操作符的表达式的等价性检测 (Tom Lane)

E.34.3.1.4. GEQO

- 每次GEQO规划一个查询时使用相同的随机种子 (Andres Freund)

  Genetic Query Optimizer (GEQO)仍然选择随机规划，它现在总是为相同的查询选择相同的随机规划， 从而给出更加一致的性能。可以修改 `geqo_seed` 试验可选择的规划。

- 改善GEQO规划选择 (Tom Lane)

  这避免了少有的错误"failed to make a valid plan"， 并且应该提升了规划速度。

E.34.3.1.5. 优化器统计信息

- 改善 `ANALYZE` 以支持继承树统计信息 (Tom Lane)

  这对于分区表特别有用。不过，当子表改变时autovacuum目前还没有自动重新分析父表。

- 当需要重新分析时，改善autovacuum 的检测 (Tom Lane)

- 改善优化器对于大于/小于比较的估计 (Tom Lane)

  当查阅对于大于/小于比较的统计信息时，如果比较值是直方图的第一个或最后一个， 使用一个索引（如果可用）抓全当前实际字段的最小或最大值。 这大大的提高了估算接近数据范围结尾的比较值的精确度， 尤其是当范围由于新数据的添加而持续变化时。

- 允许不同值数量统计的设置使用 `ALTER TABLE` (Robert Haas)

  这允许用户重载一个字段的不同值的估计数量或百分比。这个统计通常通过 `ANALYZE` 计算，但是估计可能会差些，尤其是在有非常多的行的表上。

E.34.3.1.6. 认证

- 添加对RADIUS (Remote Authentication Dial In User Service)认证的支持 (Magnus Hagander)

- 允许LDAP(Lightweight Directory Access Protocol) 认证在"search/bind"模式下操作 (Robert Fleming, Magnus Hagander)

  这允许用户先查找，然后系统使用DN (Distinguished Name) 返回给那个用户。

- 添加 `samehost` 和 `samenet` 指定到 `pg_hba.conf` (Stef Walter)

  这些分别匹配服务器的IP地址和子网地址。

- 传递信任的SSL根证书名字到客户端，这样客户端可以返回一个合适的客户端证书 (Craig Ringer)

E.34.3.1.7. 监控

- 添加客户端设置应用名 的能力，在 `pg_stat_activity` 中显示 (Dave Page)

这允许管理员表现数据流量和通过源应用寻找故障问题。

- 添加SQLSTATE选项(`%e`)到 `log_line_prefix` (Guillaume Smet)

  这允许用户通过错误代码数量编制错误和消息的统计数据。

- 以UTF16编码写入Windows事件记录 (Itagaki Takahiro)

  现在在Windows上的PostgreSQL记录消息支持多种语言。

E.34.3.1.8. 统计计数器

- 添加 `pg_stat_reset_shared('bgwriter')` 为后端写入重置集群范围的共享统计数据 (Greg Smith)

- 添加 `pg_stat_reset_single_table_counters()` 和 `pg_stat_reset_single_function_counters()` ， 允许为单独的表和函数重置统计计数器 (Magnus Hagander)

E.34.3.1.9. 服务器设置

- 允许配置参数的设置基于database/role combinations (Alvaro Herrera)

  以前，只有每个数据库和每个角色的设置是可能的，组合是不行的。 所有角色和数据库设置现在存储在新的 `pg_db_role_setting` 系统目录中。新的psql命令 `\drds` 显示了这些设置。 遗留系统视图 `pg_roles` , `pg_shadow` , 和 `pg_user` 不显示组合设置，并且因此不再完全的代表一个用户或数据库的配置。

- 添加服务器参数 `bonjour` ， 控制启用Bonjour的服务器是否通过Bonjour通知它自己 (Tom Lane)

  缺省为off，意味着它不做通知。 这允许包装程序分配启用Bonjour的建立时不用担心个别用户可能不想要这个特性。

- 添加服务器参数 `enable_material` ， 控制在优化器中物化节点的使用 (Robert Haas)

  缺省为on。为off时，优化器将不是纯粹的为性能原因添加物化节点， 尽管他们在为了正确性的需要时仍然被使用。

- 修改服务器参数 `log_temp_files` 使用千字节为缺省文件大小单位 (Robert Haas)

  以前，如果没有声明单位，这种设置解释为字节计。

- 当 `postgresql.conf` 重新加载时日志参数值的变化 (Peter Eisentraut)

  这使得管理员和安保员审计数据库设置的修改，并且也非常方便检查 `postgresql.conf` 编辑的效果。

- 正确执行自定义服务器参数的超级用户权限 (Tom Lane)

非超级用户不能再为服务器目前不知道的参数发出 `ALTER ROLE` / `DATABASE SET` 这允许服务器正确的检查仅超级用户的参数只有通过超级用户设置。以前， 允许 `SET` 并且在会话开始的时候忽略， 使得仅超级用户的自定义参数比其应该有的作用要小的多。

E.34.3.2. 查询

- 在应用 `LIMIT` 之后执行 `SELECT FOR UPDATE` / `SHARE` 处理， 这样返回的行数总是可预见的 (Tom Lane)

  以前，并发事务所做的改变会导致 `SELECT FOR UPDATE` 返回的行数出乎意料的少于它的 `LIMIT` 声明的行数。 `FOR UPDATE` 和 `ORDER BY` 的组合也会产生令人惊讶的结果， 但是可以通过在子句中放置 `FOR UPDATE` 来纠正。

- 允许传统的和SQL标准的 `LIMIT` / `OFFSET` 语法混合 (Tom Lane)

- 在窗口函数中扩展支持的框架选项 (Hitoshi Harada)

  框架现在可以以 `CURRENT ROW` 开始，并且现在支持 `ROWS` `_n_` PRECEDING/ `FOLLOWING` 选项。

- 使得 `SELECT INTO` 和 `CREATE TABLE AS` 在它们的命令标签中返回行计数到客户端 (Boszormenyi Zoltan)

  这可以节省到客户端的整个往返，允许结果计数和页码的计算不带有附加的 `COUNT` 查询。

E.34.3.2.1. Unicode 字符串

- 在 `U&` 字符串和标识符中支持Unicode代理对（段16位表现） (Peter Eisentraut)

- 在 `E'...'` 字符串中支持Unicode逃逸 (Marko Kreen)

E.34.3.3. 对象操作

- 通过定义冲刷到磁盘加速 `CREATE DATABASE` (Andres Freund, Greg Stark)

- 只允许注释在表、视图和复合类型的字段上， 不允许在其他关系类型如索引和TOAST表上 (Tom Lane)

- 允许枚举类型 的创建不包含值 (Bruce Momjian)

- 使有存储类型 `MAIN` 的字段的值保留在主堆页上， 除了不适合在一个页面上的行 (Kevin Grittner)

  以前， `MAIN` 值被迫离开到TOAST表， 直到行的大小小于页面大小的四分之一。

E.34.3.3.1. `ALTER TABLE`

- 为 `ALTER TABLE DROP COLUMN` 和 `ALTER TABLE DROP CONSTRAINT` 实现了 `IF EXISTS` (Andres Freund)
- 允许 `ALTER TABLE` 命令重写表以跳过WAL记录 (Itagaki Takahiro)

  这样的操作要么生成一个该表的新的拷贝，要么回滚，所以可以跳过WAL归档， 除非运行在连续归档模式。这样减少了I/O开支，并且提高了性能。
- 修复不是表的所有者执行 `ALTER TABLE` `_table_` ADD COLUMN `_col_` serial时的失败 (Tom Lane)

E.34.3.3.2. `CREATE TABLE`

- 添加在 `CREATE TABLE ... LIKE` 命令中拷贝 `COMMENTS` 和 `STORAGE` 字符串的支持 (Itagaki Takahiro)
- 添加拷贝 `CREATE TABLE ... LIKE` 命令中所有属性的快捷方式 (Itagaki Takahiro)
- 添加SQL标准的 `CREATE TABLE ... OF` `_type_` 命令 (Peter Eisentraut)

  这允许表的创建匹配一个现有的复合类型。附加约束和默认值可以在该命令中指定。

E.34.3.3.3. 约束

- 添加可推迟的唯一约束 (Dean Rasheed)

  这允许大量的更新，如 `UPDATE tab SET col = col + 1` ， 可靠的运行在有唯一索引或标记为主键的字段上。如果约束被指定为 `DEFERRABLE` ， 那么它将在语句的最后被检查，而不是在每行被更新之后。该约束检查也可以被推迟到当前事务的结尾， 允许这样的更新在多个SQL命令中传播。
- 增加了排除约束 (Jeff Davis)

  排除约束通过允许任意的比较操作符（不只是相等）概括唯一约束。 它们是用 `CREATE TABLE CONSTRAINT ... EXCLUDE` 子句创建的。 排除约束最常见的使用是声明字段记录不能重复，而不只是简单的不能相等。 这对于时间阶段和其他范围还有数组是有用的。 这个特性为许多日程、时间管理和科学应用加强了数据完整性的检查。
- 改善唯一约束违反的错误消息以报告导致该失败的值 (Itagaki Takahiro)

  例如，唯一约束违反现在可以报告 `Key (x)=(2) already exists` 。

E.34.3.3.4. 对象权限

- 添加使用新的 `GRANT` / `REVOKE IN SCHEMA` 子句在整个模式中改变大量权限的能力 (Petr Jelinek)

  这简化了对象权限的管理，并且使得应用数据安全更容易的利用数据库角色。

- 添加 `ALTER DEFAULT PRIVILEGES` 命令以控制稍后创建的对象的权限 (Petr Jelinek)

  这大大的简化了复杂数据库应用中对象权限的分配。可以为表、视图、序列和函数设置缺省权限。 缺省可以在每个模式的基础上或者数据库范围内分配。

- 添加用 `GRANT` / `REVOKE` 控制大对象(BLOB)权限的能力 (KaiGai Kohei)

  以前，任意数据库用户可以读取或者修改任何大对象。 现在可以赋予和撤销每个大对象的读取和写入权限， 并且追踪大对象的所有者关系。

E.34.3.4. 实用操作

- 使得 `LISTEN` / `NOTIFY` 存储在一个内存序列中等待事件，而不是在一个系统表中 (Joachim Wieland)

  这在保留事务支持和保证交付的现有特性时大幅度的提升了性能。

- 允许 `NOTIFY` 传送一个可选的"payload"字符串到监听器 (Joachim Wieland)

  这大大的提升了 `LISTEN` / `NOTIFY` 作为一个通用的事件队列系统的有用性。

- 允许在所有的数据库系统目录上 `CLUSTER` (Tom Lane)

  共享的目录仍然不能集群。

E.34.3.4.1. `COPY`

- 接受 `COPY ... CSV FORCE QUOTE *` (Itagaki Takahiro)

  现在 `*` 在 `FORCE QUOTE` 子句中可以用作"所有字段"的简写。

- 添加新的 `COPY` 语法，允许选项在圆括号内部指定 (Robert Haas, Emmanuel Cecchet)

  这允许未来的 `COPY` 选项有更大的灵活性。仍然支持旧的语法， 但是只对已经存在的选项。

E.34.3.4.2. `EXPLAIN`

- 允许 `EXPLAIN` 以XML、JSON或YAML 格式输出(Robert Haas, Greg Sabino Mullane)

  新的输出格式是容易机器可读的，支持分析 `EXPLAIN` 输出的新工具的发展。

- 添加新的 `BUFFERS` 选项报告 `EXPLAIN ANALYZE` 期间查询缓冲区的使用 (Itagaki Takahiro)

  这允许为单个查询更好的查询分析。缓冲区的使用不再在 log_statement_stats 和相关设置的输出中报告。

- 添加散列使用信息到 `EXPLAIN` 输出 (Robert Haas)

- 添加新的 `EXPLAIN` 语法，允许选项在圆括号内部指定 (Robert Haas)

  这允许未来的 `EXPLAIN` 选项有更大的灵活性。仍然支持旧的语法， 但是只对已经存在的选项。

E.34.3.4.3. `VACUUM`

- 改变 `VACUUM FULL` 重写整个表和重建它的索引， 而不是围绕紧致的空间移动个别的行 (Itagaki Takahiro, Tom Lane)

  以前的方法通常较慢，并且导致索引膨胀。请注意， 新的方法将在 `VACUUM FULL` 期间瞬间使用更多磁盘空间； 可能是正常被该表和它的索引占用的空间的两倍。

- 添加新的 `VACUUM` 语法，允许选项在圆括号内部指定 (Itagaki Takahiro)

  这允许未来的 `VACUUM` 选项有更大的灵活性。仍然支持旧的语法， 但是只对已经存在的选项。

E.34.3.4.4. 索引

- 允许索引在 `CREATE INDEX` 中通过省略索引名自动命名 (Tom Lane)

- 缺省的，多字段索引现在以所有它们的字段命名； 索引表达式字段现在基于它们的表达式命名 (Tom Lane)

- 重建共享系统目录索引现在是完全事务性并且崩溃安全了 (Tom Lane)

  以前，重建共享索引只允许在单机模式，并且在操作期间崩溃会使索引在一个比之前更糟糕的状态。

- 为GiST添加 `point_ops` 运算符类 (Teodor Sigaev)

  这个特性许可GiST索引 `point` 字段。 该索引可以用于查询的几个类型，如 `_point_` `<@` `_polygon_` （点在多边形中）。这应该使得许多PostGIS查询快很多。

- 为GIN索引创建使用红黑二叉树 (Teodor Sigaev)

  红黑树是自动平衡的。避免了非随机的输入顺序情况下的减速。

E.34.3.5. 数据类型

- 允许 `bytea` 值以十六进制符号书写 (Peter Eisentraut)

  服务器参数 `bytea_output` 控制十六进制还是传统格式用于 `bytea` 输出。当连接到PostgreSQL 9.0或更新的服务器时，Libpq的 `PQescapeByteaConn()` 函数自动使用十六进制格式。 不过，9.0以前的libpq版本将不能为更新的服务器处理十六进制格式。

新的十六进制格式将直接兼容更多使用二进制数据的应用，允许它们存储和检索， 而不用额外的转换。它比传统的格式的读取和写入也要快得多。

- 允许服务器参数extra_float_digits 增加到 `3` (Tom Lane)

  以前 `extra_float_digits` 的最大值设置是 `2` 。 在某些情况下，需要3位数准确的转储和恢复 `float4` 值。 pg_dump现在在从允许设置为3的服务器转储时将使用3的设置。

- 加强 `int2vector` 值的输入检查 (Caleb Welton)

E.34.3.5.1. 全文检索

- 在 `同义词` 字典中添加前缀支持 (Teodor Sigaev)
- 添加过滤字典 (Teodor Sigaev)

  过滤字典允许符号被修改后传送到随后的字典。

- 在邮件地址符号中允许下划线 (Teodor Sigaev)
- 为解析URL符号使用更符合标准的规则 (Tom Lane)

E.34.3.6. 函数

- 允许函数调用提供参数名，并且匹配它们到函数定义中命名的参数 (Pavel Stehule)

  例如，如果一个函数定义为接受参数 `a` 和 `b` ， 它可以用 `func(a := 7, b := 12)` 或 `func(b := 12, a := 7)` 调用。

- 支持本地特定的正则表达式 处理UTF-8服务器编码 (Tom Lane)

  本地特定的正则表达式功能包括大小写敏感匹配和本地特定的字符类。 以前，这些特性只在数据库使用一个单字节服务器编码（如LATIN1）时， 可以为非ASCII字符正确的工作。它们将仍然在多字节编码下错误行为， 除了UTF-8。

- 在 `to_char()` (`EEEE` 规范) 中添加对科学计数法的支持 (Pavel Stehule, Brendan Jurd)
- 使得 `to_char()` 遵守 `FM` （填充模式）按照 `Y` 、 `YY` 和 `YYY` 规格 (Bruce Momjian, Tom Lane)

  它早已遵从 `YYYY` 。

- 修复 `to_char()` 在Windows 上以正确的编码输出本地化的数值和货币字符串 (Hiroshi Inoue, Itagaki Takahiro, Bruce Momjian)
- 为多边形正确计算"重叠" 和"包含" (Teodor Sigaev)

多边形 `&&` (重叠)运算符以前选中只是为了查看两个多边形的边界框是否重叠。 现在它做一个更正确的检查。多边形 `@>` 和 `<@` （包含/包含于） 运算符以前选中是为了查看一个多边形顶点是否都包含在另一个中； 这对于一些非凸多边形会错误的报告"true"。现在它们检查一个多边形的所有线段都包含在另一个中。

E.34.3.6.1. 聚集

- 允许聚集函数使用 `ORDER BY` (Andrew Gierth)

  例如，现在支持： `array_agg(a ORDER BY b)` 。 这对于输入值顺序很重要的聚集是有用的， 并且消除了使用非标准的子查询确定顺序的需要。

- 多参数聚集函数现在可以使用 `DISTINCT` (Andrew Gierth)
- 添加 `string_agg()` 函数，组合值到一个字符串 (Pavel Stehule)
- 用 `DISTINCT` 调用的聚集函数，如果聚集转换函数没有被标记为 `STRICT` ， 那么现在传递NULL值 (Andrew Gierth)

  例如， `agg(DISTINCT x)` 可能传递一个NULL `x` 值到 `agg()` 。 这和非 `DISTINCT` 情况的行为更加一致。

E.34.3.6.2. 位字符串

- 为 `bit` 字符串添加 `get_bit()` 和 `set_bit()` 函数，为 `bytea` 映射这两个函数 (Leonardo F)
- 为 `bit` 字符串和 `bytea` 实现 `OVERLAY()` （替换） (Leonardo F)

E.34.3.6.3. 对象信息函数

- 添加 `pg_table_size()` 和 `pg_indexes_size()` ，提供一个更加用户友好的界面到 `pg_relation_size()` 函数 (Bernd Helmle)
- 为序列权限检查添加 `has_sequence_privilege()` (Abhijit Menon-Sen)
- 更新information_schema 视图以符合SQL:2008 (Peter Eisentraut)
- 使得 `information_schema` 视图为 `char` 和 `varchar` 字段正确的显示最大的八位字节长度 (Peter Eisentraut)
- 加速 `information_schema` 权限视图 (Joachim Wieland)

E.34.3.6.4. 函数和触发器创建

- 支持使用 `DO` 声明执行匿名代码块 (Petr Jelinek, Joshua Tolley, Hannu Valtonen)

这允许服务器端代码的执行不需要创建和删除临时函数定义。 代码可以以任意用户有权限定义函数的语言执行。

- 实现服从SQL标准的每字段触发 (Itagaki Takahiro)

  这样的触发器只有在指定的字段受到查询的影响时触发，如：出现在 `UPDATE` 的 `SET` 列表中。

- 添加 `WHEN` 子句到 `CREATE TRIGGER`， 允许控制触发器是否触发 (Itagaki Takahiro)

  虽然相同类型的检查总是可以在触发器内部执行，但是在外部的 `WHEN` 子句中做检查会有性能优势。

E.34.3.7. 服务器端语言

- 添加 `OR REPLACE` 子句到 `CREATE LANGUAGE` (Tom Lane)

  这对于可选择的安装一个并非早已存在的语言是有帮助的， 并且对于现在PL/pgSQL是默认安装的尤其有帮助。

E.34.3.7.1. PL/pgSQL服务器端语言

- 缺省安装PL/pgSQL (Bruce Momjian)

  如果管理员担心启用它会有安全或性能问题，该语言仍然可以从个别的数据库中删除。

- 改善PL/pgSQL变量名与函数的查询中使用的标识符冲突时的处理 (Tom Lane)

  缺省行为是有冲突时抛出一个错误，以便避免意外的行为。这个可以修改， 通过配置参数 `plpgsql.variable_conflict` 或者每个函数的选项 `#variable_conflict`，允许使用变量或者查询提供的字段。 在任何情况下，PL/pgSQL将不再尝试替换不符合语法规范的变量。

- 使PL/pgSQL使用主要的词法分析程序，而不是它自己的版本 (Tom Lane)

  这确保了精确的追踪主系统的细节行为，如字符串逃逸。一些用户变量细节， 如考虑保留在PL/pgSQL中的关键字的设置，已经因此而改变了。

- 避免为无效的记录引用抛出不必要的错误 (Tom Lane)

  现在只有引用实际抓取到时才抛出一个错误，而不是每当到达封闭的表达式时。 例如，许多人尝试在触发器中这样做：

  ```
  if TG_OP = 'INSERT' and NEW.col1 = ... then
  ```

  现在这实际上将如预期一样工作。

- 提高PL/pgSQL处理包含了删除字段的行类型的能力 (Pavel Stehule)
- 允许输入参数在PL/pgSQL函数内部分配值 (Steve Prentice)

  以前，输入参数被当做声明了 `CONST` 来看，所以函数的代码不能改变它们的值。 这个限制已被删除，以简化函数从其他没有利用该等价限制的DBMSes中移植。 输入参数现在像本地变量初始化为传入的值一样动作。

- 改进PL/pgSQL中的错误位置报告 (Tom Lane)
- 在PL/pgSQL中添加 `_count_` 和 `ALL` 选项到 `MOVE FORWARD` / `BACKWARD` (Pavel Stehule)
- 允许PL/pgSQL的 `WHERE CURRENT OF` 使用游标变量 (Tom Lane)
- 允许PL/pgSQL的 `OPEN` `_cursor_` FOR EXECUTE 使用参数 (Pavel Stehule, Itagaki Takahiro)

  这用一个新的 `USING` 子句完成了。

E.34.3.7.2. PL/Perl服务器端语言

- 添加新的PL/Perl函数： `quote_literal()` , `quote_nullable()` , `quote_ident()` , `encode_bytea()` , `decode_bytea()` , `looks_like_number()` , `encode_array_literal()` , `encode_array_constructor()` (Tim Bunce)
- 添加服务器参数 `plperl.on_init` ， 指定一个PL/Perl初始化函数 (Tim Bunce)

  `plperl.on_plperl_init` 和 `plperl.on_plperlu_init` 都可以用于初始化，分别针对信任的和不信任的语言。

- 在PL/Perl中支持 `END` 块 (Tim Bunce)

  `END` 块目前不允许数据库访问。

- 在PL/Perl中允许 `use strict` (Tim Bunce)

  Perl `strict` 检查也可以用新的服务器参数 `plperl.use_strict` 来全局启用。

- 在PL/Perl中允许 `require` (Tim Bunce)

  这主要是测试看看该模块是否加载上了，如果没有，产生一个错误。 不允许加载管理员没有通过初始化参数预加载的模块。

- 如果使用了Perl版本5.10或更新，则在PL/Perl中允许 `use feature` (Tim Bunce)
- 验证了PL/Perl返回值在服务器编码中可用 (Andrew Dunstan)

E.34.3.7.3. PL/Python 服务器端语言

- 在PL/Python中添加Unicode支持 (Peter Eisentraut)

如果需要，字符串自动转换从/到服务器编码。

- 在PL/Python中改善 `bytea` 支持 (Caleb Welton)

  传递到PL/Python中的 `Bytea` 值现在表示为二进制， 而不是PostgreSQL `bytea` 文本格式。包含空字节的 `Bytea` 值现在也从PL/Python中适当的输出。也改善了布尔值、整数值和浮点值的传送。

- 在PL/Python中支持arrays作为参数并且返回值 (Peter Eisentraut)
- 改善SQL域的映射为Python类型 (Peter Eisentraut)
- 添加Python 3支持到PL/Python (Peter Eisentraut)

  新的服务器端语言称作 `plpython3u` 。 不能用于Python 2服务器端语言的相同会话中。

- 改善PL/Python中的错误位置和异常报告 (Peter Eisentraut)

E.34.3.8. 客户应用程序

- 添加 `--analyze-only` 选项到 `vacuumdb` ， 不带有清理的分析 (Bruce Momjian)

E.34.3.8.1. psql

- 添加支持引用/逃逸psql 变量 的值作为SQL字符串或标识符 (Pavel Stehule, Robert Haas)

  例如， `:'var'` 将产生值 `var` ，引用并适当的逃逸为字符串常量， 而 `:"var"` 将产生它的值，引用并且逃逸为一个标识符。

- 在通过psql读取的脚本文件中， 忽略前导的UTF-8编码的Unicode字节顺序标记 (Itagaki Takahiro)

  这在客户端编码是UTF-8的时候启用。它改善了与某些编辑器的兼容性， 主要是在Windows上， 坚持插入这样的标记的那些。

- 修复 `psql --file -` 以适当的遵从 `--single-transaction` (Bruce Momjian)
- 当两个psql会话并行运行时，避免psql 的命令行历史重写 (Tom Lane)
- 改善psql的选项卡实现支持 (Itagaki Takahiro)
- 当启用 `\timing` 时显示它的输出，不管"quiet"模式 (Peter Eisentraut)

E.34.3.8.1.1. psql 显示

- 改善psql中封装字段的显示 (Roger Leigh)

  这个行为现在是缺省的。以前的格式通过使用 `\pset linestyle old-ascii` 可用。

- 允许psql通过 `\pset linestyle unicode` 使用喜好的Unicode线条特性 (Roger Leigh)

E.34.3.8.1.2. psql `\d` 命令

- 使 `\d` 显示从指定父表继承而来的子表 (Damien Clochard)

  `\d` 只显示子表的数量，而 `\d+` 显示所有子表的名字。

- 在 `\d index_name` 中显示索引字段的定义 (Khee Chin)

  定义对于表达式索引是有用的。

- 只在 `\d+` 中显示视图的定义查询，在 `\d` 中不显示 (Peter Eisentraut)

  总是包括认为过度冗长的查询。

E.34.3.8.2. pg_dump

- 使pg_dump/pg_restore `--clean` 总是删除大对象 (Itagaki Takahiro)
- 修复pg_dump，使其在启用 `standard_conforming_strings` 时适当的转储大对象 (Tom Lane)

  当转储到一个归档文件，并且然后从pg_restore生成脚本输出时， 以前的编码可能会失败。

- pg_restore现在在生成脚本输出时以十六进制格式发出大对象数据 (Tom Lane)

  如果该脚本然后输入到一个9.0以前的服务器，那么这可能会导致兼容性问题。 要绕开这个问题，直接恢复到该服务器。

- 允许pg_dump转储附加到复合类型字段的注释 (Taro Minowa (Higepon))
- 使pg_dump `--verbose` 在文本输出模式输出pg_dump和服务器版本 (Jim Cox, Tom Lane)

  这些在自定义输出模式中早已提供。

- pg_restore现在抱怨是否在开关和选项文件名后面残存任何命令行参数 (Tom Lane)

  以前，它默默地无视任何这样的参数。

E.34.3.8.3. pg_ctl

- 在系统重新启动该期间允许使用pg_ctl 安全的启动postmaster (Tom Lane)

  以前，pg_ctl的父进程错误的认为是基于旧的postmaster 锁文件运行postmaster，导致瞬间的不能启动数据库。

- 给予pg_ctl初始化数据库的能力（通过调用initdb） (Zdenek Kotala)

E.34.3.9. 开发工具

E.34.3.9.1. libpq

- 添加新的libpq函数 `PQconnectdbParams()` 和 `PQconnectStartParams()` (Guillaume Lelarge)

  这些函数类似于 `PQconnectdb()` 和 `PQconnectStart()` ，除了它们接受连接选项的数组以null结尾，而不是请求在一个字符串中提供所有选项。

- 添加libpq函数 `PQescapeLiteral()` 和 `PQescapeIdentifier()` (Robert Haas)

  这些函数返回适当引用和逃逸的SQL字符串字面值和标识符。 调用者不需要预先分配字符串结果，如 `PQescapeStringConn()` 要求。

- 添加每用户服务文件的支持(`.pg_service.conf`)， 在站点范围的服务文件之前检查 (Peter Eisentraut)

- 如果不能找到指定的libpq服务，那么适当的报告一个错误 (Peter Eisentraut)

- 在libpq中添加TCP keepalive 设置 (Tollef Fog Heen, Fujii Masao, Robert Haas)

  Keepalive设置早已在TCP连接的服务器端支持。

- 在提供可替代方式的平台上，避免额外的系统调用在libpq 中阻塞或疏通 `SIGPIPE` (Jeremy Kerr)

- 当 `.pgpass` 提供的口令未通过时， 在错误消息中提到口令的来源 (Bruce Momjian)

- 加载所有客户端认证文件中给出的SSL认证 (Tom Lane)

  改善了对间接签署的SSL认证的支持。

E.34.3.9.2. ecpg

- 添加SQLDA （SQL Descriptor Area）支持到 ecpg (Boszormenyi Zoltan)

- 添加 `DESCRIBE` `OUTPUT`] 声明到ecpg (Boszormenyi Zoltan)

- 添加一个ECPGtransactionStatus 函数以返回当前事务状态 (Bernd Helmle)

- 在ecpg Informix兼容模式中添加 `string` 数据类型 (Boszormenyi Zoltan)

- 允许ecpg没有限制的使用 `new` 和 `old` 变量名 (Michael Meskes)

- 允许ecpg在 `free()` 中使用变量名 (Michael Meskes)

- 使 `ecpg_dynamic_type()` 为非SQL3数据类型返回零 (Michael Meskes)

  以前，它返回该数据类型OID的负值。然而，这可能会与有效类型的OID混淆。

- 在早已有64位 `long` 类型的平台上支持 `long long` 类型 (Michael Meskes)

E.34.3.9.2.1. ecpg 游标

- 在ecpg的原始模式中添加超出范围游标支持 (Boszormenyi Zoltan)

  当调用了 `OPEN` 时，这允许 `DECLARE` 使用范围之外的变量。 这个工具早已存在于ecpg的Informix兼容模式中。

- 在ecpg中允许动态游标名 (Boszormenyi Zoltan)

- 允许ecpg在 `FETCH` 和 `MOVE` 中使用噪声字 `FROM` 和 `IN` (Boszormenyi Zoltan)

E.34.3.10. 构建选项

- 默认启用客户端线程安全性 (Bruce Momjian)

  线程安全全性选项可以用 `configure` `--disable-thread-safety` 禁用。

- 添加控制Linux内存不足杀手的支持 (Alex Hunsaker, Tom Lane)

  现在 `/proc/self/oom_adj` 允许禁用Linux 内存不足（OOM）杀手，推荐为postmaster禁用OOM杀手。 然后为postmaster的子进程重新启用OOM杀手就能满足需要了。 新的编译阶段选项 `LINUX_OOM_ADJ` 允许为子进程重新启用杀手。

E.34.3.10.1. Makefiles

- 新增 `Makefile` 指标 `world` 、 `install-world` 和 `installcheck-world` (Andrew Dunstan)

  这类似于现有的 `all` 、 `install` 和 `installcheck` 指标， 但是它们也构建HTML文档，构建并测试 `contrib` ， 和测试服务器端语言和ecpg。

- 添加数据和文档安装位置控制到PGXS Makefiles (Mark Cave-Ayland)

- 添加Makefile规则，作为一个HTML文件或者作为一个纯文本文件构建 PostgreSQL文档 (Peter Eisentraut, Bruce Momjian)

E.34.3.10.2. Windows

- 支持在64位Windows 上编译和在64位模式中运行 (Tsutomu Yamada, Magnus Hagander)

  这允许Windows上大的共享内存大小。

- 支持服务器构建使用Visual Studio 2008 (Magnus Hagander)

E.34.3.11. 源代码

- 在子目录树中发布预建的文档，而不是作为发布的原始码中的tar归档文件 (Peter Eisentraut)

例如，预建的HTML文档现在在 `doc/src/sgml/html/` 中；手册页同样的方式打包。

- 使服务器的词法分析器可重入 (Tom Lane)

  PL/pgSQL使用词法分析器需要这点。

- 提升内存分配的速度 (Tom Lane, Greg Stark)

- 用户定义的约束触发器现在在 `pg_constraint` 和 `pg_trigger` 中都有记录了 (Tom Lane)

  因为这个改变，`pg_constraint`.`pgconstrname` 现在是多余的，并且已经删除了。

- 添加系统目录字段 `pg_constraint`.`conindid` 和 `pg_trigger`.`tgconstrindid`， 以便更好的为约束执行记录索引的使用 (Tom Lane)

- 允许使用单个操作系统信号传递多个条件到后端 (Fujii Masao)

  这允许添加新的特性，不用管平台特定的信号条件的个数的约束。

- 提高源代码的测试覆盖率，包括 `contrib`、PL/Python和PL/Perl (Peter Eisentraut, Andrew Dunstan)

- 删除系统表引导对平面文件的使用 (Tom Lane, Alvaro Herrera)

  这提高了使用许多规则和数据库时的性能，并且消除了一些可能的故障条件。

- 为"引导"目录自动生成 `pg_attribute` 的初始内容 (John Naylor)

  这大大的简化了到这些目录的更改。

- 分裂 `INSERT`/`UPDATE`/`DELETE` 操作 `execMain.c` 的进程 (Marko Tiikkaja)

  现在在单独的ModifyTable节点中执行更新。这个更改对于未来的发展是必要的基础。

- 简化psql的SQL帮助文本的翻译 (Peter Eisentraut)

- 减少某些文件名的长度，以便在发布的原始码中的所有文件路径都小于100个字符 (Tom Lane)

  一些解压程序在长的文件路径上有问题。

- 添加新的 `ERRCODE_INVALID_PASSWORD` `SQLSTATE` 错误代码 (Bruce Momjian)

- 经过作者的许可，删除少量剩余的个人源码版权声明 (Bruce Momjian)

  个人版权声明是无关紧要的，但是社区偶尔不得不回答关于他们的问题。

- 添加新的关于在非持久模式中运行PostgreSQL以提升性能的文档 章节 (Bruce Momjian)

- 重新构造HTML文档 `Makefile` 规则， 以使它们的依赖性检查正确的工作，避免不必要的重建 (Peter Eisentraut)

- 为手册页的建立使用DocBook XSL样式表， 而不是Docbook2X (Peter Eisentraut)

  这改变了建立手册页需要的工具集。

- 改善PL/Perl代码结构 (Tim Bunce)
- 改进PL/Perl中的错误环境报告 (Alexey Klyukin)

E.34.3.11.1. 新构建的需求

请注意，从发布的原始码建立时没有应用这些需求， 因为原始码包括用于建立的这些程序的文件。

- 需要Autoconf 2.63建立configure (Peter Eisentraut)
- 需要Flex 2.5.31或更新版本从CVS检查建立 (Tom Lane)
- 需要Perl版本5.8或更新版本从CVS检查建立 (John Naylor, Andrew Dunstan)

E.34.3.11.2. 移植性

- 为Bonjour使用更现代的API (Tom Lane)

  Bonjour支持现在需要OS X 10.3或更新版本。 老的API已经被Apple弃用了。

- 为SuperH体系结构添加自旋锁支持 (Nobuhiro Iwamatsu)
- 如果支持，则允许非GCC编译器使用内联函数 (Kurt Harriman)
- 删除对没有运行64位整数数据类型的平台的支持 (Tom Lane)
- 调整 `LDFLAGS` 的使用，使其在平台间更加的一致 (Tom Lane)

  `LDFLAGS` 现在用于连接可执行文件和共享库，我们在连接可执行文件时添加 `LDFLAGS_EX` ， 在连接共享库时添加 `LDFLAGS_SL` 。

E.34.3.11.3. 服务器编程

- 使后端头文件安全的包含在C++中 (Kurt Harriman, Peter Eisentraut)

  这些改变删除关键字冲突，这些冲突在以前使得C++在后端代码中使用困难。 不过，在为后端功能使用C++时，仍然有其他复杂性。 在适当的地方仍然需要 `extern "C" { }`，内存管理和错误处理仍然是有问题的。

- 添加 `AggCheckCallContext()` ， 用于检测是否一个C函数被作为一个聚集调用 (Hitoshi Harada)
- 为 `SearchSysCache()` 和相关的函数修改调用规范， 以避免硬线连接缓存键的最大数量 (Robert Haas)

现有的调用目前仍然可以工作，但是如果没有转变到新的风格，预计会在9.1或更新的版本中打断。

- 需要调用 `fastgetattr()` 和 `heap_getattr()` 后端宏命令，以提供非NULL第四参数 (Robert Haas)
- 自定义typanalyze函数应该不再依赖于 `VacAttrStats`.`attr` 确定他们将要传送的数据的类型 (Tom Lane)

  变为允许收集索引字段上的统计信息，该索引字段的存储类型不同于底层字段的数据类型。 有新的字段告诉我们被分析的实际的数据类型。

E.34.3.11.4. 服务器 Hook

- 添加解析器hook处理ColumnRef和ParamRef节点 (Tom Lane)
- 添加处理实用程序hook，这样可加载的模块可以控制实用程序命令 (Itagaki Takahiro)

E.34.3.11.5. 二进制升级支持

- 添加 `contrib/pg_upgrade` 支持在线升级 (Bruce Momjian)

  这避免了升级到一个新的PostgreSQL主版本时转储/重载数据库的需要， 因此，通过数量级减少了停机时间。支持从PostgreSQL 8.3或8.4升级到9.0。

- 添加支持在二进制升级期间保存关系 `relfilenode` 值 (Bruce Momjian)
- 添加支持在二进制升级期间保存 `pg_type` 和 `pg_enum` OIDs (Bruce Momjian)
- 移动表空间中的数据文件到PostgreSQL版本特定的子目录中 (Bruce Momjian)

  这简化了二进制升级。

E.34.3.12. 普通发布版

- 添加多线程选项(`-j`)到 `contrib/pgbench` (Itagaki Takahiro)

  这允许pgbench使用多个CPU，降低pgbench本省成为测试障碍的风险。

- 添加 `\shell` 和 `\setshell` 元命令到 `contrib/pgbench` (Michael Paquier)
- 为 `contrib/dict_xsyn` 新增特性 (Sergey Karpov)

  新选项是 `matchorig` ， `matchsynonyms` 和 `keepsynonyms` 。

- 添加全文本字典 `contrib/unaccent` (Teodor Sigaev)

  这个过滤的字典删除了字母的重（zhong）读， 使得多种语言上的全文本搜索更加简单。

- 添加 `dblink_get_notify()` 到 `contrib/dblink` (Marcus Kempe)

  这允许dblink中的异步通知。

- 改善 `contrib/dblink` 处理删除了的字段 (Tom Lane)

  这影响了 `dblink_build_sql_insert()` 和相关的函数。这些函数现在根据逻辑而不是物理字段编号来编号字段。

- 大大的增加了 `contrib/hstore` 的数据长度限制，并且增加了B-tree和哈希支持，所以在 `hstore` 字段上 `GROUP BY` 和 `DISTINCT` 操作成为可能 (Andrew Gierth)

  也添加了新的函数和操作符。这些改进使得 `hstore` 一个功能完整键值存储嵌入到PostgreSQL中。

- 添加 `contrib/passwordcheck`， 支持场地特定密码强度政策 (Laurenz Albe)

  这个模块的源代码应该被修改为实现场地特定密码政策。

- 添加 `contrib/pg_archivecleanup` 工具 (Simon Riggs)

  目的是用于 `archive_cleanup_command` 服务器参数， 以删除不再需要的归档文件。

- 添加查询文本到 `contrib/auto_explain` 输出 (Andrew Dunstan)

- 添加缓冲区访问计数到 `contrib/pg_stat_statements` (Itagaki Takahiro)

- 更新 `contrib/start-scripts/linux`， 以使用 `/proc/self/oom_adj` 禁用Linux内存溢出(OOM)杀手 (Alex Hunsaker, Tom Lane)

E.35. 发布8.4.18

> 发布日期: 2013-10-10

该发布包含8.4.17的各种修复。关于8.4主要版本的新功能信息， 参阅Section E.53。

E.35.1. 迁移到8.4.18

运行8.4.X不需要备份/恢复。

同时，如果你从8.4.10的更早版本中升级， 参阅8.4.10的发布说明。

E.35.2. 变化

- 防止多字节编码中非ASCII非双引号标识符转换(Andrew Dunstan)

  以前的操作是错误的并且混乱的。

- 修复通过 `lo_open()` 故障导致的内存泄露(Heikki Linnakangas)

- 当 `work_mem` 正使用超过24GB内存时， 修复内存过量使用错误(Stephen Frost)

- 修复libpq SSL死锁错误(Stephen Frost)

- 正确计算包含许多NULL值的布尔列的行估计(Andrew Gierth)

  当估计计划成本时，以前的测试像 `col IS NOT TRUE` 和 `col IS NOT FALSE` 没有正确把NULL值因素计算在内。

- 防止下压 `WHERE` 子句到不安全 `UNION/INTERSECT` 子查询(Tom Lane)

  以前这种操作可能产生错误。

- 修复通过不当处理数据类型修饰符导致的特殊的 `GROUP BY` 查询错误(Tom Lane)

- 允许转储编码更好地处理基本表上已删除的列(Tom Lane)

- 修复并发 `CREATE INDEX CONCURRENTLY` 操作间的可能的死锁(Tom Lane)

- 修复零长度匹配的 `regexp_matches()` 处理(Jeevan Chalke)

  以前，零长度匹配像'^'可以返回许多匹配。

- 修复过于复杂的正则表达式崩溃(Heikki Linnakangas)

- 修复为反向引用与非贪婪量词相结合的正则表达式匹配错误(Jeevan Chalke)

- 防止 `CREATE FUNCTION` 检查 `SET` 变量 除非启用函数体检查(Tom Lane)
- 修复 `pgp_pub_decrypt()` 因此为带有密码的秘钥工作(Marko Kreen)
- 删除无索引表的清理中罕见的不正确警告(Heikki Linnakangas)
- 当在预备查询中执行事务控制命令时(比如 `ROLLBACK`)， 避免可能错误(Tom Lane)
- 允许所有平台上无穷大的各种拼写(Tom Lane)

  支持的无穷大值是"inf", "+inf", "-inf", "infinity", "+infinity"和"-infinity"
- 扩展比较行记录和数组能力(Rafal Rzepecki,Tom Lane)
- 为Israel，Morocco，Palestine，Paraguay中DST变化规律的DST变化更新 时区数据文件到tzdata发布2013d。 同时为Macquarie Island历史时区数据修正(Tom Lane)

E.36. 发布8.4.17

> 发布日期: 2013-04-04

该发布包含8.4.16的各种修复。关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.36.1. 迁移到版本8.4.17

运行8.4.X不需要备份/恢复。

然而，该发布修正了GiST索引管理中的几个错误。 安装这个更新之后，对于 `REINDEX` 满足下面描述的一个或者多个条件的任何GiST索引是明智的。

同时，如果你从8.4.10更早版本更新，参阅8.4.10的发布说明。

E.36.2. 变化

- 在每个postmaster子进程中重置OpenSSL随机状态(Marko Kreen)

  这避免了这样一种情况，其中通过 `contrib/pgcrypto` 产生的随机值 可能对于另一个数据库用户相对容易。当postmaster可以配置 `ssl` = `on` 时， 该风险是唯一显著的，但是大多数连接不使用SSL加密。(CVE-2013-1900)

- 当它不适合这样做的时候，修复GiST索引不使用"模糊"几何比较(Alexander Korotkov)

  核心几何类型执行使用"模糊"平等比较，但是 `gist_box_same` 必须执行精确比较，否则使用它的GiST索引可能变得不一致。 安装该更新之后，用户应该在 `box` ， `polygon` ， `circle` 或者 `point` 列上 `REINDEX` 任何GiST索引。 因为所有这些使用 `gist_box_same` 。

- 修复不正确的范围并集以及 为了可变宽度数据类型使用 `contrib/btree_gist` 的GiST索引中惩罚逻辑， 也就是 `text` ， `bytea` ， `bit` 和 `numeric` 列(Tom Lane)

  这些错误可能导致不一致索引， 其中一些出现的关键字不会被搜索发现， 并且在无用的索引膨胀中， 在安装此更新后建议用户 `REINDEX` 这种索引。

- 修复为多列索引在GiST页中分离代码的错误(Tom Lane)

  这些错误可能导致不一致索引， 其中一些出现的关键字不会被搜索发现， 并且在索引中是不必要的无效的搜索。 在安装此更新后建议用户 `REINDEX` 多列GiST索引。

- 修复正则表达式编译中无限循环风险(Tom Lane,Don Porter)

- 修复正则表达式编译中潜在的空指针取消引用(Tom Lane)
- 修复 `to_char()` 在合适的地方使用ASCII大小写折叠规则(Tom Lane)

  这修复了应该区域独立的一些模板模式的不当操作， 但是在Turkish地区胡乱操作" `I` "和" `i` "。
- 修复时间戳 `1999-12-31 24:00:00` 的不必要拒绝(Tom Lane)
- 删除无效的"picksplit不支持二次分裂"日志消息(Josh Hansen, Tom Lane)

  该消息似乎被添加到从未写入的期望代码中， 并且可能从来不是，因为二次分裂的GiST的缺省处理 实际上相当好。 所以停止打扰关于它的最终用户。
- 修复发送会话的 最后几个事务提交/终止计数到统计收集器的可能错误(Tom Lane)
- 消除PL/Perl的 `spi_prepare()` 函数中内存泄露(Alex Hunsaker, Tom Lane)
- 修复pg_dumpall以 正确处理包含" `=` "的数据库名字(Heikki Linnakangas)
- 当给定一个不正确连接字符串时，避免pg_dump崩溃(Heikki Linnakangas)
- 忽略pg_dump中无效索引(Michael Paquier)

  备份无效索引可能导致恢复时间的问题， 比如如果索引创建失败的原因是它试图强制 不满足表的数据的唯一性条件。同时，如果索引创建实际上仍然在进行中， 认为它是一个不受约束的DDL变化似乎是合理的，其中pg_dump 不期望备份。
- 修复 `contrib/pg_trgm` 的 `similarity()` 函数为少于三个的字符串返回零(Tom Lane)

  以前它返回 `NaN` 由于内部除以零。
- 为了Chile, Haiti, Morocco, Paraguay和一些Russian区域中DST变化规律更 新时间区域数据文件到tzdata发布2013b。 同时为更多地方修正历史区域数据。

  另外，在Russia和其他地方更新最近变化的时区缩写文件： `CHOT` ， `GET` ， `IRKT` ， `KGT` ， `KRAT` ， `MAGT` ， `MAWT` ， `MSK` ， `NOVT` ， `OMST` ， `TKT` ， `VLAT` ， `WST` ， `YAKT` ， `YEKT` 现在遵从它们当前意义，并且 `VOLT` (Europe/Volgograd)和 `MIST` (Antarctica/Macquarie)被添加到缺省缩写列表中。

E.37. 发布8.4.16

> **Release Date:** 2013-02-07

该发布包含8.4.15的各种修复。关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.37.1. 迁移到版本8.4.16

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本本更新，参阅8.4.10的发布说明。

E.37.2. 变化

- 防止来自SQL `enum_recv` 的执行(Tom Lane)

  该函数被错误声明， 允许简单SQL命令导致服务器崩溃。 原则上攻击者可以使用它检查服务器内存的内容。 我们该感谢Sumit Soni (通过Secunia SVCRP)报告这个问题。(CVE-2013-0255)

- 当截断关系文件时，更新最小恢复点(Heikki Linnakangas)

  一旦数据被丢弃，在时间轴早一点的位置停止恢复不再安全。

- 修复SQL语法以允许来自子SELECT结果的下标或者字段选择(Tom Lane)

- 当扫描 `pg_tablespace` 时，防止竞争条件(Stephen Frost, Tom Lane)

  如果有 `pg_tablespace` 项的并发更新， 那么 `CREATE DATABASE` 和 `DROP DATABASE` 可能行为不当。

- 防止 `DROP OWNED` 试图删除整个数据库或者表空间(Álvaro Herrera)

  为了安全起见，这些对象所有权必须被重新分配，而不是删除。

- 修复 `vacuum_freeze_table_age` 实现中的错误(Andres Freund)

  在安装中不只存在 `vacuum_freeze_min_age` 事务， 这个错误防止自动清理使用部分表扫描， 因此相反可能会发生全表扫描。

- 当两次解析分析 `RowExpr` 或者 `XmlExpr` 的时候， 防止不当操作(Andres Freund, Tom Lane)

  这个错误在上下文中是用户可见的， 比如 `CREATE TABLE LIKE INCLUDING INDEXES` 。

- 提高在哈希表大小计算中防御整数溢出(Jeff Davis)
- 拒绝 `to_date()` 中超期范围日期(Hitoshi Harada)
- 确保非ASCII提示符字符串被翻译成Windows上正确代码页(Alexander Law, Noah Misch)

  这个错误影响psql和一些其它客户端程序。

- 当没有连接到数据库时， 修复psql的 `\?` 命令中可能崩溃(Meng Qingzhong)
- 修复libpq的 `PQprintTuples` 中一字节缓冲区溢出(Xi Wang)

  这个旧的函数通过PostgreSQL自身并不在任何地方使用， 但是它可能被一些客户端代码使用。

- 使得ecpglib正确使用已翻译消息(Chen Huajun)
- 在MSVC上正确安装ecpg_compat和 pgtypes库(Jiang Guiqing)
- 为已提供函数重新安排配置测试， 因此它不会被假输出libedit/libreadline而欺骗(Christoph Berg)
- 确保随着时间推移Windows编译数增加(Magnus Hagander)
- 当交叉编译Windows时，使得pgxs生成正确 `.exe` 后缀的可执行文件(Zoltan Boszormenyi)
- 添加新的时区缩写 `FET` (Tom Lane)

  用于一些东欧时区。

E.38. 发布8.4.15

> 发布日期: 2012-12-06

该发布包含了8.4.14中的各种修复。 关于8.4主要发布的新功能信息，参阅Section E.53。

E.38.1. 迁移到版本8.4.15

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本开始更新，参阅8.4.10的发布说明。

E.38.2. 变化

- 修复与 `CREATE INDEX CONCURRENTLY` 有关的多个错误(Andres Freund, Tom Lane)

  当改变索引的 `pg_index` 行的状态时，修复 `CREATE INDEX CONCURRENTLY` 以使用就地更新。 这避免了竞争条件， 它可以导致并发会话错过更新目标索引， 因此导致崩溃同时已创建索引。

  同时，修复各种其他操作以确保他们忽略来源于 失败的 `CREATE INDEX CONCURRENTLY` 命令的无效索引。 这些中最重要的是 `VACUUM` ， 因为在采取的纠正措施用于修复或删除无效索引之前， 在表上自动清理可以很容易地被运行。

- 当内存不足时，避免内部哈希表崩溃(Hitoshi Harada)

- 修复外连接上非严格等价子句规划(Tom Lane)

  规划器可以从等同于其它东西的非严格建构的分句中获取不正确约束， 比如 当 `foo` 来源于外连接失效端时， `WHERE COALESCE(foo, 0) = 0` 。

- 提高规划器证明来自等价类排除约束的能力(Tom Lane)

- 修复在哈希子计划中部分行匹配以正确处理交叉类型情况(Tom Lane)

  这影响到多列 `NOT IN` 子计划，比如 `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` 。 当比如 `b` 和 `y` 分别为 `int4` 和 `int8` 。 这个错误导致不正确结果或者是依赖于具体所涉及数据类型的崩溃。

- 当为 `AFTER ROW UPDATE/DELETE` 触发器重新读取旧的元组时， 那么获取缓冲锁(Andres Freund)

在非常罕见的情况下， 这一疏忽可能导致传递不正确数据到为外键执行触发器预检查逻辑。 这可能导致崩溃， 或者关于是否触发触发器的错误决定。

- 修复 `ALTER COLUMN TYPE` 以正确处理继承的检查约束(Pavan Deolasee)

  这在8.4之前发布中正常运行， 并且在8.4以及以后发布中也正常运行。

- 修复 `REASSIGN OWNED` 以处理表空间授权(Álvaro Herrera)
- 为视图系统列忽略不正确的 `pg_attribute` 项(Tom Lane)

  视图没有任何系统列。然而，当转换表到视图时，我们忘了删除这项。 9.3以及以后的正确修复，但是在以前的分支中我们需要防御已经存在的 错误转换视图。

- 修复规则输出正确备份 `INSERT INTO` `_table_` DEFAULT VALUES (Tom Lane)
- 当在查询中有许多 `UNION` / `INTERSECT` / `EXCEPT` 子句时， 防止堆栈溢出(Tom Lane)
- 当最小的可能的整数值除以-1时，避免平台相关故障(Xi Wang, Tom Lane)
- 修复数据解析中可能访问字符串末尾(Hitoshi Harada)
- 如果Unix域套接字路径名长度超过平台特定限制， 那么会产生可以理解的错误消息(Tom Lane, Andrew Dunstan)

  先前，这可能导致一些无用的东西，比如 "域名解析中不可恢复故障"。

- 当发送复合列值给客户端时，修复内存泄露(Tom Lane)
- 使得pg_ctl关于读取 `postmaster.pid` 文件更加鲁棒(Heikki Linnakangas)

  修复竞争条件和可能的文件描述符泄露。

- 如果给出错误编码数据并且 `client_encoding` 设置是 客户端编码，比如SJIS，那么修复psql中的可能崩溃(Jiang Guiqing)
- 修复 `tar` 输出格式中通过pg_dump发出的 `restore.sql` 脚本中的错误(Tom Lane)

  该脚本可能在它的名字包含大写字符的表中完全失败。 另外，使得脚本有能力存储数据到 `--inserts` 模式和规则的COPY模式。

- 修复pg_restore以接受POSIX标准的 `tar` 文件(Brian Weaver, Tom Lane)

  pg_dump的 `tar` 输出模式的原编码产生与 POSIX标准不完全一致的文件。这是9.3版本的校正。 这个补丁更新先前分支，以致于它们接受正确的和不正确格式， 当发布9.3时，希望避免兼容性问题。

- 当给定一个数据目录的相对路径时， 修复pg_resetxlog以正确定位 `postmaster.pid` (Tom Lane)

该错误可能导致pg_resetxlog没有注意到使用数据目录的 活跃的postmaster。

- 修复libpq的 `lo_import()` 和 `lo_export()` 函数以正确报告文件I/O错误(Tom Lane)
- 修复嵌套结构指针变量的ecpg的处理(Muhammad Usama)
- 当检查页的时候，使得 `contrib/pageinspect` 的btree页检查函数 采取缓冲锁(Tom Lane)
- 修复pgxs以支持AIX上编译可加载模块(Tom Lane)

  编译不在AIX上运行的源码树外部模块。

- 为了Cuba, Israel, Jordan, Libya, Palestine, Western Samoa以及Brazil部分中的DST变化规律 更新时区数据文件到tzdata发布2012j。

E.39. 发布8.4.14

> 发布日期: 2012-09-24

该发布包含了8.4.13的各种修复，关于8.4主要发布新功能的信息， 参阅Section E.53。

E.39.1. 迁移到版本8.4.14

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本更新，参阅8.4.10的发布说明。

E.39.2. 变化

- 修复执行器参数的规划器分配， 为CTE规划节点修复执行器的重新扫描逻辑(Tom Lane)

  这些错误可以导致来自扫描相同`WITH`子查询多次的查询的错误结果。

- 提高GiST索引中页分隔决定(Alexander Korotkov,Robert Haas, Tom Lane)

  多列GiST索引可能由于这个错误遭受意外膨胀。

- 如果仍持有该权限，那么修复级联权限撤销以停止(Tom Lane)

  如果我们撤销一些角色 _X_ 的grant选项，但是 _X_ 仍然认为该选项通过其他人的grant。 我们不应该递归地撤销 _X_ 授予的角色 _Y_ 的 相应特权。

- 当使用PL/Perl的时候，修复`SIGFPE`的处理(Andres Freund)

  Perl重置进程的`SIGFPE`处理器到`SIG_IGN`， 这可能在以后导致崩溃。 在初始化PL/Perl之后恢复正常Postgres信号处理程序。

- 当被执行时，如果重新定义递归的PL/Perl函数， 则防止PL/Perl崩溃(Tom Lane)

- 解决PL/Perl中可能的错误优化(Tom Lane)

  一些Linux发布包含 导致PL/Perl中不正确编译代码的`pthread.h`不正确版本， 如果PL/Perl函数调用抛出错误的另外一个，那么导致崩溃。

- 为了Fiji中DST变化规律更新时区数据文件到tzdata发布2012f

E.40. 发布8.4.13

> 发布日期: 2012-08-17

该发布包含来自8.4.12各种修复。 关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.40.1. 迁移到版本8.4.13

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本更新，参阅8.4.10的发布说明。

E.40.2. 变化

- 通过XML实体引用避免访问外部文件/URL(Noah Misch, Tom Lane)

  `xml_parse()` 可以尝试读取外部文件夹或者URL作为需要解决DTD 以及XML值中的实体引用，从而允许未授权数据库用户尝试读取与数据库服务器权限的数据。 当外部数据还没直接返回给用户时，如果数据没有解析为有效XML，那么它的一部分可能会暴露 在错误信息中；并且无论如何检查文件是否存在的能力可能对攻击者有用。

- 阻止通过 `contrib/xml2` 的 `xslt_process()` 访问外部文件夹/URL(Peter Eisentraut)

  libxslt提供通过样式表命令读写文件夹和URL的能力， 从而允许未授权数据库用户以读写带有数据库服务器权限的数据。 禁止通过libxslt的安全选项的正确使用。(CVE-2012-3488)

  同时， 删除 `xslt_process()` 的能力从外部文件夹/URL中读取文件和样式表。 当这是已证明"特性"时，那么它被长期视为坏主意。 为了CVE-2012-3489修复打破该能力，而不是付出努力尝试修复它， 我们只是打算简单地删除它。

- 防止btree索引页过早回收利用(Noah Misch)

  当我们允许只读事务略过已分配XID时， 当只读事务仍然运行到它时， 那么我们介绍已删除btree页可以被重新利用的可能性。 这会导致不正确索引搜索结果。 在该字段产生错误的可能性很低，因为时间要求， 但尽管如此它应该被修复。

- 修复带有新创造或者重新设置序列的碰撞安全漏洞(Tom Lane)

  如果在新创造或者重新设置序列上执行 `ALTER SEQUENCE` ， 然后在它上精确执行一个 `nextval()` 调用， 然后服务器崩溃了，WAL回放可以恢复序列到似乎没有执行 `nextval()` 的状态下， 然而允许通过下一个 `nextval()` 调用再次返回第一个序列值。

特别是这可以表现为 `serial` 列，因为串行列的序列的创建包含 `ALTER SEQUENCE OWNED BY` 步骤。

- 在 `pg_start_backup()` 之后确保 `backup_label` 文件是fsync (Dave Kerr)
- 后面补丁9.1改进以压缩fsync请求队列(Robert Haas)

  在检查点期间提高了性能。 9.1变化已看到似乎安全的足够字段测试到后面补丁中。

- 仅仅允许autovacuum通过直接的封锁进程被自动取消(Tom Lane)

  原代码可能允许某些情况下的不一致操作； 特别是，在少于 `deadlock_timeout` 宽限期后可以取消autovacuum。

- 改善autovacuum取消记录(Robert Haas)
- 修复日志收集器以致于在服务器启动后 第一个日志循环期间运行 `log_truncate_on_rotation` (Tom Lane)
- 修复 `WITH` 附属于嵌套设置操作 (`UNION` / `INTERSECT` / `EXCEPT`)(Tom Lane)
- 确保参照子查询的整行不包含任何额外的 `GROUP BY` 或者 `ORDER BY` 列(Tom Lane)
- 在 `CREATE TABLE` 期间 `CHECK` 约束和索引定义中不 允许拷贝整行引用(Tom Lane)

  这种情况可以产生带有 `LIKE` 或者 `INHERITS` 的 `CREATE TABLE` 。 复制整列变量被错误地标记带有不是一个新的原来表的行类型。 为 `LIKE` 拒绝理由似乎是合理的， 因为行类型可能后面会分散。 为 `INHERITS` 我们理论上应该接受它， 伴随对父表的行类型的隐含 胁迫；但比起后端补丁似乎是安全的需要更多的工作。

- 修复 `ARRAY(SELECT ...)` 子查询中的内存泄露 (Heikki Linnakangas, Tom Lane)
- 从正则表达式修复常见前缀提取(Tom Lane)

  该代码被量化的括号子表达式搞糊涂了，比如 `^(foo)?bar` 。 这将导致这种模式不正确的搜索索引优化。

- 修复 `interval` 常量中带有解析符号 `_hh_``:``_mm_` 和 `_hh_``:``_mm_``:``_ss_` 字段错误(Amit Kapila, Tom Lane)
- 正确报告 `contrib/xml2` 的 `xslt_process()` 中错误(Tom Lane)
- 为了Morocco和Tokelau中DST变化规律更新时区数据文件到tzdata发布2012e

E.41. 发布8.4.12

> 发布日期: 2012-06-04

该发布包含8.4.11中各种修复。 关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.41.1. 迁移到版本8.4.12

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本更新， 参阅8.4.10发布说明。

E.41.2. 变化

- 修复`contrib/pgcrypto`的DES `crypt()`函数中 不正确的密码转换(Solar Designer)

  如果密码字符串包含字节值`0x80`， 那么忽略剩余的密码， 导致密码比它出现的更加弱。 使用这个修复，剩余字符串被恰当地包含在DES哈希中。 受这个错误影响的任何存储密码值将不再匹配，因此 存储值可能需要被更新。(CVE-2012-2143)

- 为程序语言的调用处理器忽略`SECURITY DEFINER`和`SET`属性(Tom Lane)

  应用这些属性到调用处理器可以使服务器崩溃(CVE-2012-2655)

- 允许`timestamp`输入 中数值时区偏移量远离UTC达到16小时(Tom Lane)

  一些历史时区有大于15小时的偏移量，先前限制。 这可能导致备份数据值在重载期间被拒绝。

- 当给定时间恰恰是当前时区的最后DST转变时间时， 修复时间戳转换处理(Tom Lane)

  这次疏忽已有很长时间，但是以前没有被注意到， 因为假设大多数DST时区有未来DST转换的不明确的序列。

- 修复`text`到`name`并且`char`到`name`投射以便在 多字节编码中正确执行字符串截断(Karl Schnaitter)

- 修复`to_tsquery()`中内存复制错误(Heikki Linnakangas)

- 修复子查询内PlaceHolderVars外的规划器处理 (Tom Lane)

  这个错误涉及到子SELECT，它引用来自周围查询的外部连接的空侧的变量。 在9.1中，这个错误影响的查询可能 伴随有"错误：在不被预期的地方发现上层PlaceHolderVar"而失败。 但是在9.0和8.4中，你可能默默地获得可能的错误结果，因为当需要时， 传递到子

查询中的值不能定位到空。

- 当 `pg_attribute` 很大的时候，修复缓慢会话启动(Tom Lane)

  如果 `pg_attribute` 超过了 `shared_buffers` 的四分之一， 在会话开始时有时需要缓存重建代码可以触发同步扫描逻辑， 导致它采取比正常更长的时间。 如果许多新会话马上开始，那么问题是相当严重的。

- 确保顺序扫描合理地检查查询取消(Merlin Moncure)

  遇到许多包含非活跃元组连续页的扫描不会同时响应中断。

- 确保返回之前 `PGSemaphoreLock()` 清除 `ImmediateInterruptOK` 的Windows实现(Tom Lane)

  这种疏忽意味着在同一个查询中后来收到的查询取消中断可能在不安全时间 被接受，伴随着不可预知的但不好的结果。

- 当输出视图或者规则时，安全显示整行变量(Abbas Butt, Tom Lane)

  涉及歧义名字(也就是说，该名字可以是一个表或者查询的列名)的情况被以模糊方式输出， 冒险转储和重载之后不同地解释视图或者规则。 通过附加无操作计算避免模凌两可的情况。

- 修复 `COPY FROM` 以正确处理与无效编码一致的空标记字符串(Tom Lane)

  一个空标记字符串比如 `E'\\0'` 应该工作， 并且工作于过去，但是这种情况在8.4中被打破。

- 确保autovacuum工作进程恰当执行堆栈深度检查(Heikki Linnakangas)

  先前，通过自动 `ANALYZE` 调用的无限递归函数可以 使工作进程崩溃。

- 修复日志收集器在高负载下没有丢失日志一致性(Andrew Dunstan)

  如果它太忙，那么收集器先前可能重新收集大的信息失败。

- 修复日志收集器以确保它在接收SIGHUP之后 重启文件旋转(Tom Lane)

- 如果索引随后被删除，那么修复GIN索引WAL回放逻辑而不失败(Tom Lane)

- 修复PL/pgSQL的 `RETURN NEXT` 命令中内存泄露(Joe Conway)

- 当该目标是函数的第一个变量时，修复PL/pgSQL的 `GET DIAGNOSTICS` 命令(Tom Lane)

- 修复psql的扩展显示(`\x`)模式中潜在的 访问内存结尾(Peter Eisentraut)

- 当数据库包含许多对象时，修复pg_dump中 若干性能问题(Jeff Janes, Tom Lane)

  如果数据库包含许多视图，或者如果许多对象在依赖循环中， 或者如果有许多拥有的序列， 那么pg_dump可能会很慢。

- 修复`contrib/dblink`的`dblink_exec()`不泄露临时数据库连接错误(Tom Lane)
- 修复`contrib/dblink`以报告错误消息中正确连接名(Kyotaro Horiguchi)
- 为了在Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria和 Tokelau Islands中DST变化规律 更新时区数据文件到tzdata发布2012c； 同时为Canada历史修正。

E.42. 发布8.4.11

> 发布日期: 2012-02-27

该发布包含8.4.10各种修复。关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.42.1. 迁移到版本8.4.11

运行8.4.X不需要备份/恢复。

然而，如果你从8.4.10更早版本更新，参阅8.4.10发布说明。

E.42.2. 变化

- 需要为 `CREATE TRIGGER` 触发器函数上的执行权限(Robert Haas)

  这个丢失检查可能允许其他用户执行带有伪造输入数据的触发器函数， 通过安装它到它拥有的表上。 对于触发器函数标记 `SECURITY DEFINER` 是唯一重要的， 因为否则触发器函数运行为表所有者。(CVE-2012-0866)

- 删除SSL证书中常见名称长度的任意限制(Heikki Linnakangas)

  libpq和服务器截断从32字节SSL证书中提取的通用名。 这通常会导致没有什么比一个意想不到的验证失败更糟糕的，但是有一些 令人难以置信的情况，它可能允许一个证书持有者模仿另外一个。 该受害者必须有32字节长的通用名。并且攻击者必须说服可信任CA发布证书， 其中通用名有字符串作为前缀。 伪装服务器也需要一些额外的开发重定向客户端连接。(CVE-2012-0867)

- 在名字写入pg_dump说明中转换新行到空白(Robert Haas)

  pg_dump关于审查输出脚本中SQL注释发出的对象名是不谨慎的。 包含换行符的名字至少使得脚本语法上不正确。 当脚本被重新加载时，恶意制作对象名可能出现SQL注射风险。(CVE-2012-0868)

- 修复来自并行清理插入的btree索引崩溃(Tom Lane)

  通过插入引起的索引页分裂有时可以导致同时运行 `VACUUM` 而错过删除本应该删除的索引项。相应表行被删除后， 该悬挂索引项可能导致错误（比如"不能读取文件中块N..."）或者更糟， 无关行后默默的错误查询结果被重新插入到当前自由表位置。 这个错误自从发布8.2就出现了， 但是发生如此罕见以致它没有被诊断直到现在。 如果你有理由怀疑它已经在你的数据库中发生， 那么重新索引受影响索引将修复这问题。

- 当改变表所有者时，更新每列权限，不仅仅每个表权限(Tom Lane)

  不这样做就意味着任何先前已授权列权限 仍然显示为已被旧的所有者授权。 这意味着既不是新所有者也不是 超级用户可以撤销目前难以寻找的到表所有者权限。

- 允许 `ALTER USER/DATABASE SET` 中 一些设置的不存在值(Heikki Linnakangas)

  允许 `default_text_search_config` ， `default_tablespace` 和 `temp_tablespaces` 被设置为不知道的名字。 这是因为它们可能在另一个数据库中已知， 该设置打算使用的地方， 或者为了表空间情况因为表空间可能不会被创建。 同样问题是先前已确认为 `search_path` ， 并且这些设置像那一个。

- 当我们删除后提交表文件有问题时，避免崩溃(Tom Lane)

  删除表导致事务提交之后删除底层磁盘文件。 在失败的情况中（比如，由于错误文件权限） 那么该代码应该发出警告信息并且继续， 因为它太晚了而终止了事务。 这个逻辑已作为发布8.4被打破， 导致这种情况引起PANIC和不可重新启动的数据库。

- 在WAL回放期间正确跟踪OID计数器，即使当它包围周围(Tom Lane)

  先前OID计数器可以保持在较高的值上直到系统退出回放模式。 实际结果通常为零，但是存在这样一种情况， 备用服务器提升到主服务器可能需要很长时间 增加OID计数器到一个合理值，一旦该值是必须的。

- 修复带有 `*` 附属的正则表达式逆向引用(Tom Lane)

  而不是执行一个确切的字符串匹配， 该代码有效地接受任何满足模式子表达式引用逆向引用符号的字符串。

  类似问题仍然困扰着被嵌入到大的量化表达式中的逆向引用，而不是量词的直接主题。 这将在未来PostgreSQL发布中得以解决。

- 修复 `inet` / `cidr` 值处理中最新引进的内存泄露(Heikki Linnakangas)

  PostgreSQL的2011年12月份发布的补丁 导致了这些操作中内存泄露，这可能是重要情况比如在这样的列上 编译btree索引。

- 修复SQL语言函数中 `CREATE TABLE AS` / `SELECT INTO` 之后的悬挂指针(Tom Lane)

  在大多数情况中这导致断言启用编译中断言失败， 但是更糟结果是可能的。

- 避免Windows上syslogger中文件句柄的双关闭(MauMau)

  通常这个错误是无形的，但是当在Windows的调试版本上运行时， 它可能导致异常。

- 修复plpgsql中I/O转换关系内存泄露(Andres Freund, Jan Urbanski, Tom Lane)

  某些操作可能泄露内存直到当前函数结束。

- 提升继承表列的pg_dump的处理(Tom Lane)

  pg_dump处理不当的情况下，一个子列 比它的父列有不同的缺省表达式。 如果缺省文本上与父类的缺省是相同的，但不是真的 相同（例如，由于模式搜索路径的差异）不会被认为是不同的， 所以在转储和恢复后 子类可以被允许继承父的缺省。在它们的父类不能微妙地错误地被恢复的地方 子列 `非空` 。

- 为了INSERT形式表数据修复pg_restore的 直接到数据库模式(Tom Lane)

  当使用发布日期2011年九月或者十二月的pg_restore的时候， 从归档文件中恢复直接到数据库伴随 `--inserts` 或者 `--column-inserts` 选项而失败， 作为另外一个问题修复的疏忽结果。 归档文件本身没有错，而且文本模式输出是好的。

- 在ecpg `DEALLOCATE` 语句 中允许 `AT` 选项(Michael Meskes)

  支持这个的基础设施已有一段时间了， 但由于疏忽仍然有拒绝这种情况的错误检查。

- 修复 `contrib/intarray` 的 `int[] &int[]` 操作符 中的错误(Guillaume Lelarge)

  如果最小整数的两个输入数组中常见的是1， 并且在其中之一数组中有较小的值，然后将1 从结果中错误地省略。

- 修复 `contrib/pgcrypto` 的 `encrypt_iv()` 和 `decrypt_iv()` 中的错误检查(Marko Kreen)

  这些函数没有成功报告无效输入错误的某些类型， 并且反而为不正确输入返回随机垃圾值。

- 修复 `contrib/test_parser` 中一字节缓冲区超出范围(Paul Guyot)

  该代码将尝试读取比它应该的又一个字节，这将在困境情况下崩溃。 因为 `contrib/test_parser` 只是示例代码， 这本身不是一个安全问题，但不好的例子代码仍然是差的。

- 如果可用，那么 在ARM上为spinlocks使用 `__sync_lock_test_and_set()` (Martin Pitt)

  这个函数替换 `SWPB` 指令先前用法，它已经废弃并且在ARMv6和之后的不可用。 报告建议该旧代码在最新ARM上以显著方式使用， 但是不能简单地连锁并发访问，导致多进程操作中的离奇失败。

- 当编译接受它的gcc版本时， 使用 `-fexcess-precision=standard` 选项(Andrew Dunstan)

  这阻止各种各样的情节，其中gcc最新版本将产生创造性结果。

- 允许FreeBSD上线程Python的使用(Chris Rees)

  我们配置脚本先前认为这种组合不会允许；但是FreeBSD修复该问题， 因此删除错误检查。

E.43. 发布8.4.10

> 发布日期: 2011-12-05

该发布包含8.4.9的各种修复， 关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.43.1. 迁移到版本8.4.10

运行8.4.X不需要备份/恢复。

然而，在 `information_schema.referential_constraints` 视图的定义中 发现了一个长期错误。如果你依赖该视图的正确结果，那么 你应该像下面第一个更新记录项解释的替换它的定义。

同时，如果你从8.4.8更早版本更新，参阅8.4.8发布说明。

E.43.2. 变化

- 修复 `information_schema.referential_constraints` 视图中错误(Tom Lane)

  该视图对于匹配依赖主键的外键约束或者唯一性约束不够仔细。 这可能导致显示所有外键约束的错误，或者显示多次，或者 声明它取决于比确实存在的不同约束。

  因为该视图定义是通过initdb安装的， 只是升级不会修复该问题。如果你需要在现有的安装中修复这个问题 ，你可以（作为一个超级用户）删除 `information_schema` 模式，然后通过 `_SHAREDIR_` /information*schema.sql* 重新创建它。 （如果你不确定 `_SHAREDIR* `在哪里，运行` pg_config --sharedir`）必须在被修复的每个数据库中重复。

- 修复GIN索引更新WAL记录的错误回放(Tom Lane)

  这可能导致在崩溃后或者热备服务器上暂时无法找到索引项， 然而，该问题可以通过索引的下一个 `VACUUM` 被修复。

- 修复 `CREATE TABLE dest AS SELECT * FROM src` 或者 `INSERT INTO dest SELECT * FROM src` 期间TOAST相关数据损坏(Tom Lane)

  如果表通过 `ALTER TABLE ADD COLUMN` 被修改， 那么尝试逐字拷贝它的数据到另一个表在某些困境情况下可以产生崩溃结果。 该问题表现在8.4以及之后版本的精确形式中， 但是我们补丁早期版本以及有其他编码路径下可以触发相同错误。

- 修复toast表访问陈旧syscache项中的竞争条件(Tom Lane)

  典型症状是短暂错误像"为pg_toast_2619中toast值NNNNN丢失块号0"， 其中引用的toast表总是从属于一个系统目录。

- 跟踪用于参数缺省表达式函数依赖(Tom Lane)

  以前，被引用的对象没有删除或者修改函数而被删除， 当使用该函数时， 导致错误操作。 请注意， 仅仅安装此更新将不能修复丢失依赖项； 这样， 你之后需要 `CREATE OR REPLACE` 每个函数。 如果你有缺省依赖非内置对象的函数， 这样做是值得推荐的。

- 允许有多个OUT参数的设置返回SQL函数的内联(Tom Lane)

- 使得 `DatumGetInetP()` 解压有1字节头的inet数据， 并且添加一个新宏， `DatumGetInetPP()` 确实没有(Heikki Linnakangas)

  这个变化不影响核心代码，但是 可能阻止希望 `DatumGetInetP()` 按惯例产生解压数据的附加代码中崩溃。

- 提高 `money` 类型的输入和输出的区域支持(Tom Lane)

  除了不支持所有标准的 `lc_monetary` 格式选项， 输入和输出函数是一致的， 意味着有区域备份 `money` 值不能被重读。

- 不要让 `transform_null_equals` 影响 `CASE foo WHEN NULL ...` 结构 (Heikki Linnakangas)

  `transform_null_equals` 只会 影响直接由用户编写的 `foo = NULL` 表达式， 通过 `CASE` 这种形式内部产生的不平等检查。

- 改变外键触发器创建顺序更好地支持自我参照外键(Tom Lane)

  一个级联外键引用它自己的表， 行更新 将触发 `ON UPDATE` 触发器和 作为一个事件的 `CHECK` 。 `ON UPDATE` 触发器必须首先执行， 否则 `CHECK` 将检查 该行的非最终状态并且可能抛出一个不合适错误。 然而， 这些触发器的触发顺序是由自己名字决定的， 其中通常按照创建顺序排序， 因为 触发器按照惯例"RI_ConstraintTrigger_NNNN" 有自动生成的名字。 一个适当的修复将需要修改该惯例， 我们会在9.2中执行， 但在现有的版本中改变它似乎有风险。 所以这个补丁只改变 触发器的创建顺序。用户遇到此类型的错误要删除并重新创建外键约束使得它的 触发器进入正确的顺序。

- 当跟踪缓冲区分配率时， 避免浮点下溢(Greg Matthews)

  当对自身无害时， 在某些平台上这可能导致讨厌的内核日志信息。

- 当在Windows下启动子进程时， 保留配置文件名字和行号值(Tom Lane)

  以前， 这些在 `pg_settings` 视图中不能被正确显示。

- 保留psql的命令历史中该命令中的空白行(Robert Haas)

  如果从字符串中删除空行， 前者操作可能产生问题， 比如。

- 修复pg_dump以备份自动生成类型之间用户定义的映射， 比如表rowtype(Tom Lane)

- 使用xsubpp首选版本以编译PL/Perl， 不一定操作系统的主拷贝(David Wheeler和 Alex Hunsaker)

- 修复 `contrib/dict_int` 和 `contrib/dict_xsyn` 中错误编码(Tom Lane)

  一些函数错误地假设通过 `palloc()` 返回的内存保证为零

- 接受 `pgstatindex()` 中的 及时查询取消中断(Robert Haas)

- 确保VPATH编译正确安装所有服务器头文件(Peter Eisentraut)

- 缩短详细错误消息中报告的文件名(Peter Eisentraut)

  规则编译一直被包含错误消息调用的C文件名报告， 但是VPATH编译之前报告绝对路径名。

- 修复中美洲Windows时区名解释(Tom Lane)

  映射"中美洲标准时间"为 `CST6` ， 而不是 `CST6CDT` ， 因为在中美洲任何地方通常观察不到DST。

- 为了Brazil, Cuba, Fiji, Palestine, Russia和Samoa中DST变化规律 更新时区数据文件到tzdata发布2011n； 以及历史修正Alaska和British East Africa。

E.44. 发布8.4.9

> 发布日期: 2011-09-26

该发布包含来自8.4.8的各种修复。关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.44.1. 迁移到版本8.4.9

为运行8.4.X不需要备份/恢复。

然而，如果你从8.4.8更早版本更新，参阅8.4.8发布说明。

E.44.2. 变化

- 修复存在问题的热更新元组的索引错误(Tom Lane)

  这些错误可以导致重新索引系统目录后索引崩溃。 他们不认为会影响用户索引。

- 修复GiST索引页分裂处理的多个错误(Heikki Linnakangas)

  发生的概率是很低的，但是可以导致索引失败。

- 修复 `tsvector_concat()` 中可能的缓冲区溢出(Tom Lane)

  该函数低估它的结果所需要的内存量，导致服务器崩溃。

- 当处理"standalone"参数时，修复 `xml_recv` 中崩溃(Tom Lane)

- 使得 `pg_options_to_table` 为没有值的选项返回NULL(Tom Lane)

  以前这种情况可以导致服务器崩溃。

- 避免在 `ANALYZE` 和SJIS-2004编码转换中可能访问内存结尾(Noah Misch)

  这修复了一些概率很低的服务器崩溃情况。

- 防止间歇性挂在启动进程和bgwriter进程的相互作用中(Simon Riggs)

  这影响了在非热备份情况中的恢复。

- 修复relcache初始文件失效的竞态条件(Tom Lane)

  有一个Window，其中新的后台进程可以读一个陈旧的初始化文件， 但是忽略了告知它的数据是陈旧的无效消息。 其结果在目录访问中是奇怪的错误，通常在启动之后 "无法读取文件中块0..."。

- 修复在GiST索引扫描结尾的内存泄露(Tom Lane)

  执行许多单独GiST索引扫描的命令， 比如包含很多行的表上新的基于排斥约束的GiST验证， 由于这种泄露可能短暂地需要大量内存。

- 修复元组存储可支持游标以及 plpgsql的 `RETURN NEXT` 命令中不正确的内存计算（可能导致内存膨胀）(Tom Lane)

- 当建立一个大的，有损耗的位图时，修复性能问题(Tom Lane)

- 修复唯一列的连接选择性估计(Tom Lane)

  这修复了可以导致连接结果大小的较差估计的错误的规划器探试。

- 修复只出现在子select目标列中的嵌套PlaceHolderVar表达式(Tom Lane)

  这个错误可以导致错误的出现空的外部连接的输出。

- 运行正确优化的嵌套的 `EXISTS` 查询(Tom Lane)

- 修复数组和路径创建函数确保填充字节为零(Tom Lane)

  这避免了规划器认为语义上相同的常数是不相等的一种情况，导致低劣的优化。

- 修复 `EXPLAIN` 以处理内部索引扫描子规划中控制结果节点(Tom Lane)

  这种忽视的通常状况是"bogus varno"错误。

- 解决了打破WAL回放的gcc 4.6.0错误(Tom Lane)

  这可能会导致服务器崩溃后已提交事务损失。

- 修复视图中 `VALUES` 备份错误(Tom Lane)

- 不允许序列上 `SELECT FOR UPDATE/SHARE` (Tom Lane)

  该操作不按预期运行并且导致错误。

- 修复 `VACUUM` 所以它总是更新 `pg_class` . `reltuples` / `relpages` (Tom Lane)

  这将修复对于当vacuum表的时候autovacuum可能会越来越差的一些情况。

- 当计算哈希表大小时，防止整数溢出(Tom Lane)

- 修复 `CLUSTER` 可能尝试访问已删除TOAST数据(Tom Lane)

- 修复为"peer"认证使用凭证控制消息的可能错误(Tom Lane)

- 当需要多次往返时，修复SSPI登录(Ahmed Shinwari,Magnus Hagander)

  这个问题典型症状是在SSPI登录期间"不支持该函数请求"错误。

- 如果 `pg_hba.conf` 包含 `hostssl`，但是 SSL被禁用，抛出错误(Tom Lane)

  这一结论比默默忽略这些行的先前操作更加人性化。

- 修复 `pg_srand48` 起源初始化中的typo (Andres Freund)

  这导致错误使用已提供的种子的所有位。 在大多数平台上不使用这个功能（只有那些没有 `srandom` 的）， 以及来自任何情况下比预期更少随机种子似乎最小的潜在安全隐患。

- 当 `LIMIT` 和 `OFFSET` 总数值超过2^63的时候， 避免整数溢出(Heikki Linnakangas)
- 添加溢出检查到 `generate_series()` 的 `int4` 和 `int8` 版本(Robert Haas)
- 修复 `to_char()` 中尾随零删除(Marti Raudsepp)

  带有 `FM` 的格式，并且小数点后没有数字位置中， 小数点左边的零可能被错误地删除。

- 修复 `pg_size_pretty()` 以避免接近2^63的输入溢出(Tom Lane)
- 削弱记录值中typmod匹配的plpgsql检查(Tom Lane)

  一个过分热情检查可能导致丢弃一直保持的长度修饰符。

- 正确处理initdb期间的区域名的引用(Heikki Linnakangas)

  这种情况可以产生一些Windows区域，比如"中华人民共和国"。

- 从8.3更新期间修复pg_upgrade保存toast表的relfrozenxid (Bruce Momjian)

  如果不这样做可能导致 `pg_clog` 文件升级后很快被删除。

- 在pg_ctl中， 支持Windows上服务注册静止模式(MauMau)
- 修复从不同文件 `COPY` 时脚本文件行数的psql计数(Tom Lane)
- 为 `standard_conforming_strings` 修复 pg_restore的直接到数据库模式(Tom Lane)

  当从 `standard_conforming_strings` 设置为 `on` 的归档文件中直接恢复 到数据库服务器时，pg_restore可以发出错误命令。

- 关于平行的pg_restore不支持情况用户界面友好(Tom Lane)

  该变化确保了在采取的任何恢复操作之前这种情况被检测和报告。

- 修复写入超越缓冲区末尾以及在libpq的LDAP服务查找代码中的内存泄漏(Albe Laurenz)
- 在libpq中，当使用非阻塞I/O和SSL连接时避免错误(Martin Pihlak, Tom Lane)
- 连接启动时提高libpq的错误处理(Tom Lane)

  特别是，在SSL连接启动时 `fork()` 错误的服务器报告 的反应是明智的。

- 提高SSL故障libpq的错误报告(Tom Lane)
- 当添加新元组到最初从服务器查询获得的 `PGresult` 时， 修复 `PQsetvalue()` 避免可能崩溃(Andrew Chernow)
- 使得ecpglib写入带有15位数字精度的 `double` 值(Akira Kurosawa)
- 在ecpglib中，确保错误之后恢复 `LC_NUMERIC` 设置(Michael Meskes)
- 提供上游修复加密的符号字符错误(CVE-2011-2483)(Tom Lane)

  `contrib/pg_crypto` 的加密代码可以在 字符是有符号的（这是大多数）平台上提供错误结果，导致加密密码比它们所应该的更弱。

- 修复 `contrib/seg` 中的内存泄露(Heikki Linnakangas)
- 修复 `pgstatindex()` 为空索引提供一致结果(Tom Lane)
- 允许编译 perl 5.14 (Alex Hunsaker)
- 为探测系统函数的存在更新配置脚本方法(Tom Lane)

  在8.3和8.2中使用的autoconf的版本可能被执行链接时优化的编译器愚弄。

- 修复包含空格的编译安装文件路径的相关问题(Tom Lane)
- 更新时区数据文件到为Canada, Egypt, Russia, Samoa和South Sudan中的DST变化规律的tzdata发布2011i。

E.45. 发布8.4.8

> 发布日期: 2011-04-18

该发布包含来自8.4.7的各种修复， 关于8.4主要发布的新功能信息， 参阅Section E.53。

E.45.1. 迁移到版本8.4.8

运行8.4.X不需要备份/恢复。

然而，如果你的安装是通过运行pg_upgrade从 原先主要发布中更新，你应该采取措施避免 由于pg_upgrade中现在修复错误 可能的数据丢失。建议方法是在所有TOAST表上运行 `VACUUM FREEZE` 。 http://wiki.postgresql.org/wiki/20110408pg_upgrade_fix中更多信息可用。

另外，如果你从8.4.2更早版本更新，参阅8.4.2的发布说明。

E.45.2. 变化

- 修复pg_upgrade的TOAST表处理(Bruce Momjian)

  TOAST表的 `pg_class` . `relfrozenxid` 值 不能被正确拷贝到pg_upgrade中的新安装中。 当它们仍然需要验证TOAST表中的元组时， 这可能会导致丢弃 `pg_clog` ， 导致"无法访问事务状态"错误。

  这个错误引起使用pg_upgrade更新的安装中数据丢失的显著风险。 该补丁校对pg_upgrade未来使用中的错误， 但是自身并不能解决已被pg_upgrade旧版本处理的安装中的问题。

- 抑制不正确的"PD_ALL_VISIBLE标记被不正确地设置"警告(Heikki Linnakangas)

  `VACUUM` 有时在实际有效的情况下发出警告。

- 不允许完全包含复合类型(Tom Lane)

  当处理复合类型的时候，这可以避免服务器无限地重现的情况。虽然有些可能 用这种结构，它们似乎不足以证明可以确保安全工作。

- 避免目录缓存初始化期间潜在死锁(Nikhil Sontakke)

  在某些情况下，缓存加载代码会在锁定索引的目录之前的系统索引上获取共享锁。 这可能停顿过程以试图获取排他锁，采用更多的标准顺序。

- 当有一个并发更新到目标元组时，修复 `BEFORE ROW UPDATE` 触发器处理中的 悬挂指针问题(Tom Lane)

  当尝试执行 `UPDATE RETURNING ctid` 的时候， 已被观察到的这个错误导致间歇性"无法从虚拟元组提取系统属性"错误。 对于更严重的错误有一个非常小的概率，如为更新的元组产生不正确的索引项。

- 当有表的等待延迟触发器事件时，不允许 `DROP TABLE` (Tom Lane)

  以前 `DROP` 可以通过，当触发器最终被唤起时， 导致"无法打开OID nnn关系"的错误。

- 在GEQO优化期间，避免通过常量错误WHERE条件触发崩溃(Tom Lane)

- 提高半连接和反连接情况中规划器的处理(Tom Lane)

- 修复解释空的原因的文本搜索的选择性估计(Jesper Krogh)

- 提高处理已删除列行类型的PL/pgSQL的能力(Pavel Stehule)

  这是在9.0之前修复后补丁。

- 修复涉及到数组片段的PL/Python内存泄露(Daniel Popowich)

- 修复pg_restore处理TOC文件中长线(超过1KB) (Tom Lane)

- 采取更多防御措施阻止崩溃，由于使用过度热情的编译器优化除以零(Aurelien Jarno)

- 支持MIPS上FreeBSD和OpenBSD中dlopen()的使用(Tom Lane)

  有一个硬连线假设，即该系统函数 在这些系统上的MIPS硬件上不可用。代替使用编译测试， 因为最新版本中存在。

- 修复HP-UX上编译失败(Heikki Linnakangas)

- 解决Windows上与libintl版本不兼容问题(Hiroshi Inoue)

- 修复Windows编译脚本中xcopy的使用 以便在Windows 7中正常工作(Andrew Dunstan)

  这只影响到编译脚本，没有安装或者使用。

- 修复Cygwin上的pg_regress使用的路径分隔符(Andrew Dunstan)

- 为了Chile, Cuba, Falkland Islands, Morocco, Samoa, 和Turkey中DST变化规律更新时区数据文件到tzdata发布2011f； 同时为South Australia, Alaska和Hawaii进行历史修正。

E.46. 发布8.4.7

> 发布日期: 2011-01-31

该发布包含8.4.6各种修复。关于8.4主要发布的新功能的各种信息， 参阅Section E.53。

E.46.1. 迁移到版本8.4.7

运行8.4.X不需要备份/恢复。然而，如果你从8.4.2更早版本更新， 参阅8.4.2的发布说明。

E.46.2. 变化

- 当 `EXPLAIN` 尝试显示简单形式 `CASE` 表达式时， 避免错误(Tom Lane)

  如果 `CASE` 的测试表达式是常量，规划器可以简化 `CASE` 到困惑表达式显示代码形式，导致"不希望的CASE WHEN子句"错误。

- 修复分配给下标的现有范围的数组片段(Tom Lane)

  如果在新添加的下标和最早预先存在的下标之间存在差距， 该代码误算了许多需要从旧的数组无效的位图拷贝的项， 可能导致数据损坏或崩溃。

- 在规划非常远的日期值时避免意外转换溢出(Tom Lane)

  `date` 类型支持比通过 `timestamp` 类型表示的更广泛日期， 但是规划器假设它完全可以将日期转换为timestamp。

- 当 `standard_conforming_strings` 是on时， 为大对象(BLOBs)修复pg_restore的 文本输出(Tom Lane)

  虽然直接存储到数据库正常工作，如果 pg_restore要求SQL文本输出并且 `standard_conforming_strings` 在源数据库中启用，字符串逃逸是不正确的。

- 修复包含 `... & !(subexpression) | ...` 的 `tsquery` 值的错误解析(Tom Lane)

  包含操作符这种组合的查询不能正确被执行。同样的错误 存在于 `contrib/intarray` 的 `query_int` 类型和 `contrib/ltree` 的 `ltxtquery` 类型中。

- 修复 `query_int` 类型的 `contrib/intarray` 的输入函数中 的缓冲区溢出(Apple)

  这个错误是一个安全风险，因为该函数的返回地址可以被覆盖。 感谢 Apple Inc的安全团队报告这个问题并且提供修复(CVE-2010-4015)

- 修复 `contrib/seg` 的GiST picksplit算法中的错误(Alexander Korotkov)

这可能导致相当大的无效率，虽然在`seg`列上的GiST索引中不是真的错误答案。 如果你有这样一个索引，可以在安装这个更新之后考虑下`REINDEX`它。 （这与以前更新中`contrib/cube`中被修复的错误是相同的）。

E.47. 发布8.4.6

> 发布日期: 2010-12-16

该发布包含8.4.5的各种修复。 关于8.4主要发布的新功能的信息， 参阅Section E.53。

E.47.1. 迁移到版本8.4.6

运行8.4.X不需要备份/恢复。 然而，如果你从8.4.2更早版本更新， 参阅8.4.2发布说明。

E.47.2. 变化

- 强迫缺省 `wal_sync_method` 为Linux上 `fdatasync` (Tom Lane, Marti Raudsepp)

  Linux上缺省实际上是多年的 `fdatasync` ， 但是最近内核变化导致 PostgreSQL选择 `open_datasync` 。 这种选择在任何性能改进中不产生结果，并且导致某些文件系统的彻底失败， 尤其是带有 `data=journal` 挂载选项的 `ext4` 。

- 修复GIN索引WAL回放逻辑相关错误(Tom Lane)

  这可能导致"坏缓冲区id: 0"失败 或者复制过程中索引内容的损坏。

- 当开启检查点WAL记录不在同一WAL段作为重做点时， 修复基础备份恢复(Jeff Davis)

- 当多个执行者保持活跃很长时间的时候，修复autovacuum的持续放缓(Tom Lane)

  为autovacuum工作者有效的 `vacuum_cost_limit` 可能下降到几乎为零， 如果它处理足够多的表，使其运行极其缓慢。

- 添加支持 `IA64` 上的检测寄存器栈溢出(Tom Lane)

  `IA64` 结构有两个硬件堆栈。堆栈溢出错误的充分预防需要检查这两个。

- 添加 `copyObject()` 中栈溢出检查(Tom Lane)

  某些代码路径可能导致崩溃，由于堆栈溢出产生了一个足够复杂的查询。

- 修复临时GiST索引中页分离的检测(Heikki Linnakangas)

  当执行插入时，如果比如有一个打开游标扫描索引的时候，在临时索引中可能 有"并发"的页面分离。当继续执行游标时， GiST未能检测到这种情况，因此可以提供错误结果。

- 在早期连接处理过程中修复错误检查(Tom Lane)

在一些情况下忽略太多子进程检查，当尝试添加新的子进程到固定大小数组中时， 可能导致postmaster崩溃。

- 提升window函数的效率(Tom Lane)

  某些情况中大量元组需要提前被读取，但是 `work_mem` 大到足以允许它们 全部保存在内存中，竟意外地慢。 `percent_rank()` , `cume_dist()` 和 `ntile()` 尤其受到这个问题的影响。

- 当 `ANALYZE` 复杂索引表达式的时候，避免内存泄露(Tom Lane)

- 确保使用整行Var的索引依赖于它的表(Tom Lane)

  当删除表的时候，像 `create index i on t (foo(t.*))` 声明的索引 可能不会自动被删除。

- 不要"内联"SQL函数与多个 `OUT` 参数(Tom Lane)

  由于关于预期结果rowtype信息丢失，这避免了可能的崩溃。

- 如果 `ORDER BY` , `LIMIT` , `FOR UPDATE` , or `WITH` 附属于 `INSERT ... VALUES` 的 `VALUES` 部分，那么操作正确(Tom Lane)

- 修复 `COALESCE()` 表达式的常数合并(Tom Lane)

  规划器有时会试图评估事实上不可能达到的可能会导致意外错误的子表达式。

- 当连接接受失败时（ `accept()` 或者在它之后立即调用），并且postmaster和GSSAPI支持一起编译， 修复postmaster崩溃。(Alexander Chernikov)

- 当 `log_temp_files` 是活跃的，则修复临时文件忽略的取消操作(Tom Lane)

  当试图发出日志信息时，如果发生错误，那么取消不执行， 导致临时文件积累。

- 为 `InhRelation` 节点添加输出功能(Tom Lane)

  当启用 `debug_print_parse` 并且执行某些类型查询时， 这避免失败。

- 修复从一个点到水平线段距离的错误计算(Tom Lane)

  这个漏洞影响了若干个不同的几何距离测量操作符。

- 修复ecpg中事务状态的错误计算(Itagaki Takahiro)

- 在递归或者错误恢复的情况中修复"simple"表达式的PL/pgSQL处理而不失败(Tom Lane)

- 修复设置返回函数的PL/Python的处理(Jan Urbanski)

  在迭代器生成一组结果中尝试调用SPI函数可能失败。

- 修复 `contrib/cube` 的GiST picksplit算法中的错误(Alexander Korotkov)

这可能会导致相当大的无效率，但在 `cube` 列上的GiST索引中，并不是真的错误结论，如果你有这样一个索引，考虑在安装这个更新之后 `REINDEX` 它。

- 不要发出 `contrib/dblink` 中"标识符将被截断"的信息除了 创建新的连接(Itagaki Takahiro)
- 修复 `contrib/pgcrypto` 中对失踪公钥的潜在信息转储(Marti Raudsepp)
- 修复 `contrib/xml2` 的XPath查询函数中的内存泄露(Tom Lane)
- 为Fiji和Samoa中的DST变化规律更新时区数据文件到tzdata发布2010o； 也为了香港历史修正。

E.48. 发布8.4.5

> 发布日期: 2010-10-04

该发布包含来自8.4.4中的各种修复。 关于8.4主要版本的新功能的各种信息，参阅Section E.53。

E.48.1. 迁移到版本8.4.5

运行8.4.X不需要备份/恢复。然而，如果你从8.4.2的早期版本升级， 参阅8.4.2的发布说明。

E.48.2. 变化

- 为每个调用PL/Perl和PL/Tcl中SQL userid使用独立解释器(Tom Lane)

  这种变化可以防止通过颠覆随后在同一会话中另一个SQL用户身份下执行的Perl或Tcl代码造成的安全问题 （例如，在 `SECURITY DEFINER` 函数中）。 大多数的脚本语言提供了可以执行的众多方法， 如重新定义标准函数或目标函数运算符。没有这种变化的话，任何拥有Perl或Tcl语言使用权的SQL用户 可以执行拥有目标函数所有者的SQL权限应该做的事情。

  这种变化的成本是在Perl和Tcl函数之间沟通意图变得更加困难。 为了提供一个逃生出口，PL/PerlU和PL/TclU函数继续每个会话中仅仅使用一个解释器。 这没有考虑安全问题，因为在数据库超级用户的信任级别上执行所有这些函数。

  可能声称提供可信执行程序的第三方程序语言有类似的安全问题。 我们建议你依赖安全关键用途联系任何PL的作者。

  我们应该感谢Tim Bunce提出这个问题(CVE-2010-3433)。

- 防止 `pg_get_expr()` 中可能的事故， 通过禁止适合与不是一个系统目录列之一的参数一起使用。(Heikki Linnakangas, Tom Lane)

- 处理退出代码128(`ERROR_WAIT_NO_CHILDREN`)作为Windows上非致命的(Magnus Hagander)

  在高负荷的情况下，Windows进程在使用错误码启动过程失败。 之前postmaster把这个看作恐慌条件并且重新启动整个数据库，但这似乎是反应过度。

- 修复占位符赋值不正确位置(Tom Lane)

当它们本应是空的时候，这个错误可能导致查询输出非空。 在该情况下，输出列表非严格的表达式下外部连接的内部是一个子选择。

- 修复 `UNION ALL` 成员关系可能的重复扫描(Tom Lane)
- 修复"不能处理非计划子查询"错误(Tom Lane)

  当包含连接别名参考的子查询扩展到含有另一个子选择的表达式中时，发生这种情况。
- 修复引用视图或出现在嵌套子查询中的子选择的整列变量的不当处理(Tom Lane)
- 修复交叉类型 `IN` 比较的处理不当(Tom Lane)

  如果规划器试图实现使用分类然后唯一的完全连接计划的 `IN` 连接。
- 修复统计 `tsvector` 列的 `ANALYZE` 的计算(Jan Urbanski)

  最初代码可以产生不正确统计，导致之后错误规划选择。
- 优化通过 `array_agg()`， `string_agg()` 和类似聚合函数使用的内存的规划器估计(Hitoshi Harada)

  以往过低估计可以导致内存不足错误，由于哈希聚集规划不恰当的选择。
- 修复标记缓存计划是短暂的错误(Tom Lane)

  当 `CREATE INDEX CONCURRENTLY` 在进展中作为一个参考表，如果已准备好一个规划， 一旦索引是现成的，它应该重新被规划。这是不会发生的。
- 在一些偶然报道btree失败的案例中减少PANIC到ERROR， 并提供产生错误信息中的额外的细节。

  这可以优化使用错误索引的系统的鲁棒性。
- 修复GIN索引部分匹配查询的错误搜索逻辑(Tom Lane)

  涉及一些GIN索引条件的AND/OR组合情况不总是给出正确答案， 有时候比必要情况慢的多。
- 防止autovacuum进程中show_session_authorization()的崩溃(Tom Lane)
- 并不是所有返回行都是同一rowtype的地方防范函数返回集合记录(Tom Lane)
- 修复在子事务回滚期间待定触发器事件列表的可能损坏(Tom Lane)

  这可能导致崩溃或者错误触发触发器。
- 当散列按引用传递函数结果时，修复可能错误(Tao Ma, Tom Lane)
- 优化join列中NULL的合并连接处理(Tom Lane)

如果排序顺序是NULL排序高，那么合并连接可以完全停止到达第一个NULL。

- 当写入它们时，注意fsync锁文件内容（ `postmaster.pid` 和socket锁文件）(Tom Lane)

  如果在postmaster启动之后不久该机器崩溃，那么该忽略可能导致损坏的锁文件内容。反过来又可以防止后续尝试启动postmaster，直到手动删除锁定文件。

- 当分配XID给严重嵌套子事务时，避免递归(Andres Freund, Robert Haas)

  如果有有限的堆栈空间，初始编码可能导致崩溃。

- 避免在walwriter过程中保持开放旧的WAL段(Magnus Hagander, Heikki Linnakangas)

  先前编码可以防止删除不再需要部分。

- 修复 `log_line_prefix` 's `%i` 逃逸，这在 后台启动早期时会产生垃圾(Tom Lane)

- 避免为TOAST表部分指定的关系选项的曲解(Itagaki Takahiro)

  特别是， `fillfactor` 会被读为零， 如果任何其他reloption已经为该表设置，导致严重的膨胀。

- 修复 `ALTER TABLE ... ADD CONSTRAINT` 中继承计数跟踪(Robert Haas)

- 当启用归档时，修复 `ALTER TABLE ... SET TABLESPACE` 中可能的数据损坏(Jeff Davis)

- 允许 `CREATE DATABASE` 和 `ALTER DATABASE ... SET TABLESPACE` 被查询取消中断(Guillaume Lelarge)

- 改善 `CREATE INDEX` 对被推荐索引表达式是否可以改变的检测(Tom Lane)

- 修复 `REASSIGN OWNED` 以处理算子类和族(Asko Tiidumaa)

- 当比较两个空的 `tsquery` 值的时候，修复可能的核心转储(Tom Lane)

- 修复模式中包含 `%` 伴随 `_` 的 `LIKE` 的处理(Tom Lane)

  我们之前已经修复，但是仍存在一些不正确处理的情况。

- 重新允许Julian日期输入追溯到0001-01-01 AD (Tom Lane)

  在8.4之前输入比如 `'J100000'::date` ，但是通过添加的错误检查无意打破。

- 如果在遍历游标的 `FOR` 循环中关闭游标，修复PL/pgSQL抛出一个错误，不会崩溃。

- 在PL/Python中，抵御来自 `PyCObject_AsVoidPtr` 和 `PyCObject_FromVoidPtr` 的空指针结果(Peter Eisentraut)

- 在libpq中，当指定 `host` 和 `hostaddr` 的时候，修复 完整SSL证书验证。

- 使得psql将 `DISCARD ALL` 看做在自动提交关闭模式中不应该封装在事务块中的命令(Itagaki Takahiro)

- 修复SQL/MED对象pg_dump的处理中的一些问题(Tom Lane)

  值得注意的是，如果通过非超级用户执行，pg_dump总是失败， 它并不打算这样。

- 优化非可查找归档文件的pg_dump和pg_restore的处理(Tom Lane, Robert Haas)

  对于并行转储的正常运作很重要。

- 提高并行pg_restore处理选择性转储(`-L` option)的能力(Tom Lane)

  如果 `-L` 文件命令非缺省转储顺序，那么原代码可能失败。

- 修复ecpg正确处理来自 `RETURNING` 子句的数据(Michael Meskes)

- 修复ecpg中的一些内存泄露(Zoltan Boszormenyi)

- 优化包含已删除列的表的 `contrib/dblink` 的处理(Tom Lane)

- 修复 `contrib/dblink` 中"重复连接名"错误之后的连接泄露(Itagaki Takahiro)

- 修复 `contrib/dblink` 以正确处理超过62字节的连接名(Itagaki Takahiro)

- 添加 `hstore(text, text)` 函数到 `contrib/hstore` (Robert Haas)

  该函数是为了现在不使用的 `=>` 操作符的推荐替代者。 它后面打补丁，以便适应未来代码可用于旧的服务器版本。 请注意，该补丁仅仅在安装 `contrib/hstore` 或 重新安装在一个特定的数据库中有效。 用户可能更愿意手动执行 `CREATE FUNCTION` 命令来代替。

- 更新基础设施建设和文档以反映来自CVS到Git的源代码存储库的移动(Magnus Hagander and others)

- 为了Egypt和Palestine中DST变化规律更新时区数据文件到tzdata发布2010l； 也是出于Finland的历史修正。

  这种变化还为两个Micronesian时区添加了新的名字：Pacific/Chuuk目前优于Pacific/Truk（并且首选缩写是CHUT而不是TRUT） 并且Pacific/Pohnpei优于Pacific/Ponape。

- 使得Windows的"N. Central Asia Standard Time"时区映射到Asia/Novosibirsk，而不是Asia/Almaty (Magnus Hagander)

  Microsoft改变来自KB976098的时区更新中的该区域的DST操作。 Asia/Novosibirsk是对于新操作的更好匹配。

E.49. 发布8.4.4

> 发布日期: 2010-05-17

该发布包含来自8.4.3的各种修复。关于8.4主要版本中的新特性的信息， 请参阅Section E.53。

E.49.1. 迁移到版本8.4.4

运行8.4.X不需要备份/恢复。然而，如果你从8.4.2更早版本更新，参阅8.4.2的发布说明。

E.49.2. 变化

- 强制 `plperl` 中使用opmask限制应用于整个解释器， 而不是使用 `Safe.pm` (Tim Bunce, Andrew Dunstan)

  最近发展使我们确信 `Safe.pm` 太不安全而不能依靠 `plperl` 可信赖。 此变化删除了 `Safe.pm` 的使用， 有利于使用应用于操作码掩码的一个单独的解析器。 该变化副作用包括以 `plperl` 中一种自然的方式使用Perl的 `strict` 编译是可能的， 并且Perl的 `$a` 和 `$b` 变量按预期以排序程序执行，而且该函数编译显著更快。

- 阻止PL/Tcl执行来自 `pltcl_modules` 的不信任代码(Tom)

  从一个数据库表中自动加载的Tcl代码的PL/Tcl的功能可以为了特洛伊木马攻击被开发，因为对谁可以创建或插入该表是没有限制的。这种变化禁用该功能， 除非 `pltcl_modules` 是由超级用户拥有。 （然而，不检查该表上的权限，所以真正需要低于安全模块表中的安装仍然可以授予适当的权限给值得信赖的非超级用户）。 另外，防止加载代码到非限制"normal" Tcl解释器，除非我们真的要执行 `pltclu` 函数。

- 修复 `ALTER ... SET TABLESPACE` 的WAL回放期间的数据损坏(Tom)

  当 `archive_mode` 是on时， `ALTER ... SET TABLESPACE` 生成WAL记录，其中 回放逻辑是不正确的。它可以将数据写入错误的地方，从而导致可能的不可恢复的数据损坏。 如果在提交 `ALTER` 之后和下一个检查点之前发生数据库崩溃和恢复， 数据损坏在备库上随时被观察，并且可能在主库上发生。?????

- 如果在relcache项重建期间收到缓存复位信息，那么修复可能崩溃(Heikki)

  当修复相关错误的时候，在8.4.3中介绍这些错误。

- 当为了该函数运行语言验证器的时候，应用每个函数GUC设置(Itagaki Takahiro)

这避免了失败，如果该函数的代码没有该设置是无效的; 一个例子是，如果 `search_path` 不正确，SQL函数可能不被解析。

- 当 `constraint_exclusion` = `partition` 的时候，为继承 `UPDATE` 和 `DELETE` 目标表执行约束排除(Tom)

  由于疏忽，该设置以前造成约束排除在 `SELECT` 命令中被检查。

- 不允许未经授权的用户重置超级用户唯一的参数设置(Alvaro)

  以前，如果一个非特权用户为自身运行 `ALTER USER ... RESET ALL`， 或为所拥有的数据库运行 `ALTER DATABASE ... RESET ALL`， 这将删除用户或数据库的所有特殊参数设置，????? 甚至是那些只应该由超级用户改变的设置。现在， `ALTER` 将只删除该用户有权限改变的参数。

- 当 `CONTEXT` 附加物为日志项时，如果发生宕机，避免后台关机时可能的崩溃(Tom)

  在某些情况中上下文输出函数可能失败，因为当它打印日志信息的时候，当前事务已经回滚。

- 修复 `recovery_end_command` 中 `%r` 参数的错误处理(Heikki)

  该值总是零。

- 确保归档进程尽可能快地响应 `archive_command` 中变化(Tom)
- 当case表达式是一个没有返回行的查询时，修复pl/pgsql的 `CASE` 语句而不失败(Tom)
- 更新pl/perl的 `ppport.h` 为现代Perl版本(Andrew)
- 修复pl/python中各种内存泄露(Andreas Freund, Tom)
- 正确处理ecpg中空字符串连接参数(Michael)
- 当扩展引用自身的一个变量时，避免在psql中的无限递归(Tom)
- 修复psql的 `\copy` 而在 `\copy (select ...)` 中点周围 不添加空格(Tom)

  在数值文字中小数点周围的空格可能导致语法错误。

- 当在上下文环境中运行不匹配 `client_encoding` 的时候， 避免psql中格式错误(Tom)
- 为了使用 `contrib/intarray` 操作符的不满足条件的查询， 修复不必要的"GIN索引不支持全局索引扫描"错误(Tom)
- 确保 `contrib/pgstattuple` 函数反应及时取消中断(Tatsuhito Kasahara)
- 确保服务器启动正确处理 `shmget()` 为已存在的共享内存段 返回 `EINVAL` (Tom)

在包括OS X中的BSD衍生的内核上观察这种操作， 这导致了抱怨共享内存请求大小过大的完全误导启动错误。

- 避免Windows上syslogger处理过程中可能的崩溃(Heikki)
- 更有力地处理Windows注册表中不完整的时区信息(Magnus)
- 更新已知的Windows时区名字的设置(Magnus)
- 为了Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia中DST变化规律更新 时区数据文件到tzdata发布2010j。 同时为Taiwan历史修正。

  另外，添加 `PKST` (Pakistan Summer Time)到时区缩写的缺省设置中。

E.50. 发布8.4.3

> 发布日期: 2010-03-15

该发布包含了来自8.4.2的各种修复。关于8.4主要版本的新功能信息，参阅 Section E.53。

E.50.1. 迁移到版本8.4.3

运行8.4.X不需要备份/恢复。然而，如果你从8.4.2更早版本更新，参阅8.4.2发布说明。

E.50.2. 变化

- 添加新的配置参数 `ssl_renegotiation_limit` 用来控制 我们多久一次执行会话秘钥重新协商SSL连接(Magnus)

  这可以被设置为零以完全禁用重新协商，如果使用一个已破碎的SSL库。这可能被要求。特别是，一些供应商为导致重新协商尝试失败的CVE-2009-3555打补丁。

- 修复后台启动过程中可能的死锁(Tom)
- 由于在relcache明确重载中没有处理错误，修复可能的崩溃(Tom)
- 由于使用悬空指针到缓存计划，修复可能崩溃(Tatsuo)
- 由于缓存计划 `ROLLBACK` 的严重失效，修复可能崩溃(Tom)
- 当试图从子事务启动故障恢复时，修复可能的崩溃(Tom)
- 修复与使用保存点和来自服务器编码不同的客户端编码有关的服务器内存泄漏(Tom)
- 修正GIST索引页拆分中结束恢复清理期间发出的不正确WAL数据(Yoichi Hirai)

  如果在结束恢复清理中已经完成了一个不完整的GIST插入后我们运气不好而发生崩溃。这将导致索引损坏，甚至在WAL回放期间更有可能发生错误。

- 为GIN索引修复WAL重做清理方法中的错误(Heikki)
- 修复GIN索引搜索中扫描键的不正确比较(Teodor)
- 使得为 `bit` 类型的 `substring()` 将任何负的长度看作 "所有剩余字符串"(Tom)

  先前代码只处理-1，并且可能为其他负值产生无效结果值，可能导致崩溃(CVE-2010-0442)。

- 当输出比特宽度大于不同于8位倍数的给定整数的时候，修复整数到比特串转换用来正确地处理第一个分数字节(Tom)

- 修复病理上阻碍正则表达式匹配的一些情况(Tom)

- 当尝试内联返回一组包含已删除列的复合类型的SQL函数的时候，修复产生的错误(Tom)

- 修复试图更新复合类型数组列元素字段中的错误(Tom)

- 当 `EXPLAIN` 必须打印FieldStore或者分配ArrayRef表达式的时候， 避免错误(Tom)

  这种情况已经出现 `EXPLAIN VERBOSE` 试图输出计划节点目标列。

- 在未修饰字符串出现在 `UNION` / `INTERSECT` / `EXCEPT` 子查询中的情况中， 避免不必要强制错误(Tom)

  修复8.4之前执行情况中的回归分析。

- 在整行Var中有包含已删除列的rowtype的地方，避免不符合要求的rowtype兼容检查错误(Tom)

- 当最后位置正好在一段边界时，修复备份历史文件中的 `STOP WAL LOCATION` 项以 报告下一个WAL段的名称(Itagaki Takahiro)

- 总是将系统目录ID传递给 `CREATE FOREIGN DATA WRAPPER` 中指定的选项 验证函数(Martin Pihlak)

- 修复临时文件泄露的一些情况(Heikki)

  这纠正了以前次要版本引入的问题。 一种失败情况是：当在另一个函数的异常处理程序中调用plpgsql函数返回集。

- 添加支持执行 `FULL JOIN ON FALSE` (Tom)

  这可以防止来自8.4之前发布的一些查询的回归，该查询现在可以简化为一个恒定的错误连接条件。

- 优化布尔变量情况下约束排除处理， 特别是有可能排除一个有"bool_column = false"约束的分区(Tom)

- 防止将 `INOUT` 映射看作表示二进制兼容(Heikki)

- 当警告不能授权或者撤销列级别权限的时候，包含该消息的字段名(Stephen Frost)

  当 `REVOKE` 产生多条消息时，这个原先似乎是复制品，这比起以前来是非常有用的， 并且避免混淆。

- 当读取 `pg_hba.conf` 和相关文件的时候，如果 `@` 似乎出现在引号内， 不要将 `@something` 看作文件包含请求， 另外，也不要将 `@` 自身看作文件包含请求(Tom)

如果角色或数据库的名字以 `@` 开头，这可以防止不稳定操作。 如果你需要包含路径名含有空格的文件，那么你仍然可以这样做，但是你必须写 `@"/path to/file"` 而不是把引号放入整个构造中。

- 如果目录被命名为 `pg_hba.conf` 中和相关文件的包含目标，则 防止在一些平台上的无限循环(Tom)
- 如果没有设置 `errno` 时 `SSL_read` 或者 `SSL_write` 失败， 修复可能无限循环(Tom)

  据说这个可能使用openssl的一些Windows版本。
- 不允许本地连接中GSSAPI认证，因为它需要hostname以正常工作(Magnus)
- 保护ecpg应用意外地释放字符串(Michael)
- 如果连接断开，使得ecpg报告合适SQLSTATE(Michael)
- 修复psql `\d` 输出中单元格内容的翻译(Heikki)
- 修复psql的 `numericlocale` 选项而不能格式化字符串， 它不属于latex和troff输出格式(Heikki)
- 修复psql中小的每个查询内存泄露(Tom)
- 当指定 `ON_ERROR_STOP` 和 `--single-transaction` 的时候，并且 在隐含 `COMMIT` 期间发生错误时，使得psql返回正确退出状态(3)(Bruce)
- 修复外部服务器权限的pg_dump的输出(Heikki)
- 由于溢出依赖性ID，在平行pg_restore中修复可能崩溃(Tom)
- 在复合列设置为空的情况下，修复plpgsql错误(Tom)
- 当从PL/PerlU中调用PL/Perl函数的时候，修复可能失败，反之亦然(Tim Bunce)
- 在PL/Python中添加 `volatile` 标记以避免可能具体编译器操作不当(Zdenek Kotala)
- 确保PL/Tcl完全初始化Tcl解释器(Tom)

  这种监督的唯一已知现象是如果使用Tcl 8.5或者更高版本，则Tcl `clock` 命令行为不当。
- 避免在失败事务或者子事务中创建入口运行 `ExecutorEnd` (Tom)

  当使用 `contrib/auto_explain` 的时候，这是已知错误。
- 当许多关键列被指定为 `dblink_build_sql_*` 函数时，阻止 `contrib/dblink` 崩溃(Rushabh Lathia, Joe Conway)
- 允许 `contrib/ltree` 操作中零维数组(Tom)

这种情况曾经被认为一个错误，但它更方便 将它看作零元素数组。当 `ltree` 操作应用于 `ARRAY(SELECT ...)` 结果和没有返回 行的子查询的时候，特别是这可以避免不必要的故障。

- 修复通过内存管理造成的 `contrib/xml2` 中各种崩溃(Tom)
- 使得 `contrib/xml2` 编译在Windows上更加具有鲁棒性(Andrew)
- 修复Windows上信号处理竞争条件(Radu Ilie)

  这个错误已知现象是在 `pg_listener` 中的行可以在重负载下被删除。

- 如果C编译器不提供64位整数数据类型，使得配置脚本报告错误(Tom)

  这种情况一段时间已经被打断，不再值得支持，因此在配置时间拒绝它。

- 为了Bangladesh，Chile，Fiji， Mexico，Paraguay，Samoa中DST变化规律更新 时区数据文件到tzdata发布2010e。

E.51. 发布8.4.2

> 发布日期: 2009-12-14

该发布包含来自8.4.1的各种修复。关于8.4主要发布中的新特性信息， 参阅Section E.53。

E.51.1. 迁移到版本8.4.2

运行8.4.X不需要备份/恢复。然而，如果有任何散列索引，在更新到8.4.2之后你应该`REINDEX`它们，以修复可能受损。

E.51.2. 变化

- 防止通过索引函数改变会话局部状态引起的间接安全威胁(Gurjeet Singh, Tom)

  这一变化防止不变的索引函数打断超级用户会话(CVE-2009-4136)。

- 拒绝在普通名字（CN）字段中包含嵌入的空字节的SSL证书(Magnus)

  这避免在SSL认证期间来自服务器或者客户端名字的证书的意外匹配(CVE-2009-4034)。

- 修复散列索引损坏(Tom)

  该8.4变化使得散列索引保持通过散列值未能更新大量分裂和紧缩程序以保持排序的项。所以这些操作的任何应用程序可能导致索引的永久性损坏，在这个意义上， 搜索可能无法找到存在的项。为了处理这个，推荐你已安装此更新后`REINDEX`任何散列索引。

- 修复后台启动时缓存初始化期间可能崩溃(Tom)

- 避免空的同义词词典崩溃(Tom)

- 避免在不安全时间中断`VACUUM`信号(Alvaro)

  如果已经坚定元组动作之后取消了`VACUUM FULL`， 该修复阻止PANIC。 如果在截断该表之后中断普通的`VACUUM`， 阻止瞬态错误。？

- 由于在哈希表大小计算中整数溢出，修复可能崩溃(Tom)

  这可能发生大的规划器评估哈希连接结果。

- 如果在内部依赖对象上尝试`DROP`， 修复崩溃(Tom)

- 修复`inet` / `cidr`比较中稀少崩溃现象(Chris Mikkelson)

- 确保不忽略通过预备事务持有的共享元组级别锁(Heikki)

- 修复用于在子事务中访问的游标临时文件的提早删除(Heikki)
- 当旋转到新的CSV日志文件时，修复syslogger处理中内存泄露(Tom)
- 当重新解析 `pg_hba.conf` 的时候，修复postmaster中的内存泄露(Tom)
- 修复Windows权限下降逻辑(Jesse Morris)

  这修复了数据库在Windows上启动失败的一些情况， 通常带有误导性错误信息比如"不能定位匹配postgres可执行程序"。

- 使得在主查询中 `FOR UPDATE/SHARE` 不扩大到 `WITH` 查询中(Tom)

  比如，在

  ```
  WITH w AS (SELECT * FROM foo) SELECT * FROM w, bar ... FOR UPDATE
  ```

  `FOR UPDATE` 将影响 `bar` 而不是 `foo` 。 这比原来8.4操作更有用而且更加一致， 尝试扩散 `FOR UPDATE` 到 `WITH` 查询中， 但是却总是失败，因为匹配实现限制。它也遵循设计规则，如果主查询独立，执行 `WITH` 查询。

- 修复在另外一个中使用 `WITH RECURSIVE` 查询的错误(Tom)
- 修复散列索引的并发错误(Tom)

  并发插入可能导致索引扫描瞬时报告错误结果。

- 当分割取决于索引的非首列时，修复GiST索引页分割错误逻辑(Paul Ramsey)
- 修复为多列GIN索引使用 `fastupdate` 激活的错误搜索结果(Teodor)
- 修复GIN索引WAL项创建中的错误(Tom)

  当 `full_page_writes` 为on的时候，这些错误被隐藏，但是如果在下一个检查点之前发生崩溃， 那么关闭WAL回放错误是一定的。

- 如果在检查点结尾回收或者删除旧的WAL文件失败，不要出错误(Heikki)

  更好地将这个问题作为非致命性的并且允许完成检查点。未来的检查点将重试取消。 这样的问题没有预期的正常运行，但通过错误设计Windows杀毒和备份软件造成的。

- 确保Windows上WAL文件不再被重复归档(Heikki)

  如果其他过程干扰不再需要的文件的删除，这可能发生另外一种情况。

- 修复PAM密码处理更加鲁棒性(Tom)

上面的代码伴随着Linux `pam_krb5` PAM模块与作为域控制器的Microsoft动态目录结合而失败。 它可能还有其他问题，因为它采用关于PAM堆栈传递给它什么参数的不合理假设。

- 提高GSSAPI和SSPI认证方法中最大认证令牌(Kerberos ticket)大小(Ian Turner)

  当为Unix Kerberos 实现有足够多的旧的2000字节限制时， 通过Windows 域控制器发出标签可以更大。

- 确保域约束在 `ARRAY[...]::domain` 结构中被强制，其中域不在数值类型中(Heikki)

- 修复为涉及复合类型列作为外键的情况中的外键逻辑(Tom)

- 确保游标的快照在被创建后不被修改(Alvaro)

  如果在同一事务中后续操作修改数据，这可能导致游标传递错误结果，应该返回游标。

- 修复 `CREATE TABLE` 以正确合并来自不同继承父表的缺省表达式(Tom)

  这用于运行，但在8.4中被损坏。

- 重新启用序列访问统计收集(Akira Kurosawa)

  这用于运行，但在8.3中被损坏。

- 修复 `CREATE OR REPLACE FUNCTION` 中所有依赖的处理(Tom)

- 修复 `WHERE``_x_` = `_x_` 条件的错误处理(Tom)

  在某些情况下，可以作为冗余忽略，但是它们不是— 它们等价于 `_x_` `IS NOT NULL` 。

- 当为文本相同易变表达式使用哈希聚合实现 `DISTINCT` 的时候， 修复不正确规划结构(Tom)

- 修复易变 `SELECT DISTINCT ON` 表达式的断言错误(Tom)

- 修复 `ts_stat()` 而在空的 `tsvector` 值上不失败(Tom)

- 使得文本搜索解析器接受XML属性中下划线(Peter)

- 修复 `xml` 二进制输入中编码处理(Heikki)

  如果XML头没有指定编码，我们现在假设缺省UTF-8；先前处理是不一致的。

- 修复从 `plperlu` 调用 `plperl` 的错误，反之亦然(Tom)

  从内部函数退出的错误可能导致崩溃，由于该错误为外部函数重新选择正确的Perl解释器。

- 当重新定义PL/Perl函数的时候，修复会话存在期内存泄露(Tom)

- 当通过集合返回PL/Perl函数返回时，确保Perl数组正确转换为PostgreSQL数组(Andrew Dunstan, Abhijit Menon-Sen)

  这些正常运行都为了非集合返回函数。

- 修复PL/Python中异常处理的罕见崩溃(Peter)
- 修复 `DECLARE CURSOR` 语句中注释的ecpg问题(Michael)
- 修复ecpg不把最近添加的关键字作为保留字(Tom)

  这影响了关键字 `CALLED` , `CATALOG` , `DEFINER` , `ENUM` , `FOLLOWING` , `INVOKER` , `OPTIONS` , `PARTITION` , `PRECEDING` , `RANGE` , `SECURITY` , `SERVER` , `UNBOUNDED` 和 `WRAPPER` 。

- 重新允许psql的 `\df` 函数名字参数中的正则表达式特殊字符(Tom)
- 在 `contrib/fuzzystrmatch` 中，纠正非缺省成本 `levenshtein` 距离计算(Marcin Mank)
- 在 `contrib/pg_standby` 中，禁用Windows上带有信号触发器转移(Fujii Masao)

  这没有什么有用的，因为Windows没有Unix风格讯号，但是最近变化确实崩溃。

- 将 `FREEZE` and `VERBOSE` 选项以正确顺序放置在 `contrib/vacuumdb` 产生的 `VACUUM` 命令中(Heikki)
- 当 `contrib/dblink` 遇到错误的时候，修复链接的可能泄露(Tatsuhito Kasahara)
- 确保psql的flex模块与正确系统标题定义一起被编译(Tom)

  这将修复平台上的编译错误，其中 `--enable-largefile` 在产生代码中导致的 不兼容变化。

- 使得postmaster忽略任何连接请求包中的 `application_name` 参数， 以优化与未来libpq版本的兼容(Tom)
- 更新时区缩写文件以匹配当前实际情况(Joachim Wieland)

  这包含添加 `IDT` 到缺省时区缩写设置。

- 为了Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria中DST变化规律更新时区数据文件到 tzdata发布2009s。也为了Hong Kong历史纠正。

E.52. 发布8.4.1

> 发布日期: 2009-09-09

该发布包含来自8.4中的各种修复。关于8.4主要发布中的新特性信息， 参阅Section E.53。

E.52.1. 迁移到版本8.4.1

运行8.4.X不需要备份/恢复。

E.52.2. 变化

- 在归档恢复结尾修复WAL页面头部初始化(Heikki)

  这可能导致在后续归档恢复中处理WAL的错误。

- 修复"归档中不能产生新的WAL项"错误(Tom)

- 修复在崩溃后使得过期行可见的问题(Tom)

  涉及页面状态位的错误可能在服务器崩溃后不能正确设置。

- 在安全定义函数中不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` (Tom, Heikki)

  这包含了在之前补丁中漏掉的一种情况， 在安全定义函数中不允许 `SET ROLE` 和 `SET SESSION AUTHORIZATION` (参阅CVE-2007-6600)

- 使得已经加载模块的 `LOAD` 成为非操作的(Tom)

  以前 `LOAD` 尝试卸载并且重新加载该模块，但是这不安全而且没有用处。

- 使得窗口函数 `PARTITION BY` 和 `ORDER BY` 项总是被解释为 简单表达式(Tom)

  在8.4.0中这些列表按照用于顶级 `GROUP BY` 和 `ORDER BY` 列表规则被解析。 但是每个SQL标准是不正确的，并且可能导致循环。

- 修复半连接规划中若干错误(Tom)

  在 `IN` 或者 `EXISTS` 与另外一个连接一起使用的情况中导致错误查询结果。

- 修复引用外连接中子查询的整行处理(Tom)

  例子为 `SELECT COUNT(ss.*) FROM ... LEFT JOIN (SELECT ...) ss ON ...` 。 这里，为空扩展连接行 `ss.*` 被看作为 `ROW(NULL,NULL,...)` ， 与简单空不一样，现在将它看作为简单空。

- 修复Windows共享内存分配代码(Tsutomu Yamada, Magnus)

  该错误导致经常报道的"不能附加到共享内存"错误信息。

- 修复本地处理plperl (Heikki)

  当调用plperl函数的时候，导致数据损坏，该错误可能导致服务器的本地设置改变。

- 修复reloptions的处理以确保设置一个选项不强制为其它的缺省值(Itagaki Takahiro)

- 确保"fast shutdown"请求强制终止打开会话，即使"smart shutdown"已经 在进行中(Fujii Masao)

- 避免 `GROUP BY` 查询中 `array_agg()` 的内存泄露(Tom)

- 将 `to_char(..., 'TH')` 看作带有 `'HH'` / `'HH12'` 后缀的大写字母序列(Heikki)

  它曾作为 `'th'` (小写字母)被处理。

- 包含 `time` 和 `time with time zone` 输出的 `EXTRACT(second)` 和 `EXTRACT(milliseconds)` 结果中的小数部分(Tom)

  一直为浮点datetime配置工作，但在整数datetime代码中被打断。

- 当 `_x_` 大于2百万并且整数datetime在使用中的时候， 修复 `INTERVAL '``_x_` ms'溢出(Alex Hunsaker)

- 当处理索引扫描中toasted值得时候，优化性能(Tom)

  这对PostGIS是特别有用的。

- 修复禁用 `commit_delay` 错误(Jeff Janes)

- 如果以silent模式启用服务器，输出早期启用信息到 `postmaster.log` 中(Tom)

  以前丢弃这样的错误信息，导致调度困难。

- 删除翻译常见问题(Peter)

  现在主要是在wiki上面。 主要的问题前段时间已经移到了wiki上。

- 如果 `postgresql.conf` 为空，那么修复pg_ctl 不陷入无限循环中(Jeff Davis)

- 修复pg_dump的 `--binary-upgrade` 模式中的若干错误(Bruce, Tom)

  通过pg_migrator使用 `pg_dump --binary-upgrade` 。

- 修复 `contrib/xml2` 的 `xslt_process()` 以正确处理 最大数量参数(二十)(Tom)

- 提高libpq代码的鲁棒性以恢复 `COPY FROM STDIN` 中错误(Tom)

- 当安装两个库的时候，避免包含readline和editline头文件的冲突(Zdenek Kotala)

- 解决gcc错误，它导致一些平台上"浮点异常"而不是"以零做除数"(Tom)
- 为Bangladesh, Egypt, Mauritius中DST变化规律更新时区数据文件到tzdata发布2009l。

E.53. 发布8.4

> 发布日期: 2009-07-01

E.53.1. 概要

经过多年研发后，PostgreSQL已经在许多领域特性比较完整。 该发布显示一个针对性方法来添加特性（比如，认证， 监测，空间重利用）， 并且添加在以后SQL标准中定义的能力。增强的主要方面是：

- 窗口函数
- 通用表表达式和递归查询
- 函数缺省和可变参数
- 并行恢复
- 列权限
- 每个数据库本地设置
- 改进的哈希索引
- 改进 `EXISTS` 和 `NOT EXISTS` 查询连接性能
- 容易使用热备份
- 空闲空间映射的自动大小
- 可见映射（大大降低了缓慢变化表的清理开销）
- 版本感知psql(旧服务器之前使用反斜杠命令)
- 支持用户认证的SSL证书
- 每个函数运行时统计
- psql中函数简单编辑
- 新的contrib模块：pg_stat_statements, auto_explain, citext, btree_gin

在下面章节中将详细解释以上所列项。

E.53.2. 迁移到版本8.4

对于那些希望从任何以前发布中迁移数据是需要使用pg_dump备份/恢复。

观察下面的不兼容：

E.53.2.1. 普遍的

- 缺省使用64位整数日期时间(Neil Conway)

  以前通过configure的 `--enable-integer-datetimes` 选项进行选择。为保留旧操作，与 `--disable-integer-datetimes` 一起编译。

- 删除ipcclean实用命令(Bruce)

  该功能仅仅工作于几个平台上。用户应该使用他们的操作系统工具。

E.53.2.2. 服务器设置

- 为 `log_min_messages` 更改缺省设置为 `warning` （以前为 `notice` ） 以减少日志文件数量(Tom)
- 为 `max_prepared_transactions` 改变缺省设置为零（以前为5）(Tom)
- 使得 `debug_print_parse` ， `debug_print_rewritten` ，和 `debug_print_plan` 在 `LOG` 消息级别输出，而不是像从前的 `DEBUG1` (Tom)
- `debug_pretty_print` 缺省为 `on` (Tom)
- 删除 `explain_pretty_print` 参数（不再需要的）(Tom)
- 通过超级用户使得 `log_temp_files` 可定位，像其它日志选项(Simon Riggs)
- 当没有 `%` 逃逸出现在 `log_filename` 中的时候，删除epoch时间戳的自动追加(Robert Haas)

  该变化是因为有些用户想要一个已修复的日志文件名，用于外部日志轮换工具。

- 从 `recovery.conf` 删除中删除 `log_restartpoints` ； 代替使用 `log_checkpoints` (Simon)
- 删除 `krb_realm` 和 `krb_server_hostname` ；而是在 `pg_hba.conf` 中设置(Magnus)
- 在 `pg_hba.conf` 中有显著变化，如下所述。

E.53.2.3. 查询

- 改变 `TRUNCATE` 和 `LOCK` 以适用于指定表的子表(Peter)

  这些命令现在接受避免处理子表的 `ONLY` 选项；如果需要旧操作，必须使用该选项。

- `SELECT DISTINCT` 和 `UNION` / `INTERSECT` / `EXCEPT` 不再总是产生有序输出(Tom)

  之前，这些类型的查询总是通过排序/唯一的处理方式（即排序然后删除相邻重复）去除重复行。 现在，他们可以通过哈希来实现，这不会生成有序输出。如果应用程序依赖于按照排序顺序的输出， ??????那么推荐解决方法是添加一个 `ORDER BY` 子句。作为一个短期解决方法， 之前的操作可以通过禁用 `enable_hashagg` 被恢复，但是这是一个性能昂贵的修复。 `SELECT DISTINCT ON` 从不使用散列，然而，因此，其操作不改变。

- 强迫子表继承父表 `CHECK` 约束(Alex Hunsaker, Nikhil Sontakke, Tom)

  以前可能从子表中删除该约束，当扫描父表的时候，允许违背该约束的行可见。 这被认为是不一致的，也违背了SQL标准。

- 禁止负的 `LIMIT` 或者 `OFFSET` 值，而不是将它们视为零(Simon)

- 禁止事务块外的 `LOCK TABLE` (Tom)

  这样的操作是无效的，因为该锁将被立即释放。

- 现在序列包含一个额外的 `start_value` 列(Zoltan Boszormenyi)

  支持 `ALTER SEQUENCE ... RESTART` 。

E.53.2.4. 函数和操作符

- 采用 `numeric` 零放到分数幂上返回 `0` ，而不是抛出错误， 并且采用 `numeric` 零放到零幂上返回 `1` ，而不是错误(Bruce)

  这匹配长期的 `float8` 操作。

- 允许浮点值的一元负号产生负零(Tom)

  修改后操作更加遵循IEEE-标准。

- 如果转义字符是 `LIKE` 模式中的最后一个字符，抛出一个错误（即它没有任何逃逸）(Tom)

  此前，这样的转义字符被默默忽略掉，从而可能掩盖应用程序逻辑错误。

- 删除以前用于 `LIKE` 索引比较的 `~=~` and `~<>~` 操作符(Tom)

  模式索引目前使用规则平等操作符。

- `xpath()` 现在将参数传递给libxml没有任何变化(Andrew)

  这意味着XML参数必须是一个良好的XML文档。先前编码试图允许XML片段， 但它并没有很好地工作。

- 让 `xmlelement()` 格式属性值就好像目录值(Peter)

此前，属性值是按照正常的SQL输出操作被格式化，有时不符合XML规则。

- 为libxml-使用函数重写内存管理(Tom)

  这种变化应该避免在PL/Perl中使用libxml和其他附加代码的一些兼容性问题。

- 为哈希函数采取更快速算法(Kenneth Marshall,依据Bob Jenkins的工作)

  许多内置哈希函数现在在小端和大端平台上提供不同的结果。

E.53.2.4.1. 时间函数和操作符

- `DateStyle` 不再控制 `interval` 输出格式； 反而有一个新变量 `IntervalStyle` (Ron Mayer)
- 提高 `timestamp` 和 `interval` 输出中小数秒处理的一致性(Ron Mayer)

  这可能导致比以前显示不同数量的小数位，或者四舍五入而不是截断。

- 使得 `to_char()` 的本地的月份/日期名字依赖于 `LC_TIME` ， 而不是 `LC_MESSAGES` (Euler Taveira de Oliveira)
- 导致 `to_date()` 和 `to_timestamp()` 更加一致报告无效输入错误(Brendan Jurd)

  以前的版本中会常常忽略或默默地误读不匹配格式字符串的输入。 这种情况下将导致错误。

- 为了子午线(`AM` / `PM`)和纪元 (`BC` / `AD`)格式名称修复 `to_timestamp()` 而不需要大写/小写匹配(Brendan Jurd)

  比如，输入值 `ad` 匹配格式字符串 `AD` 。

E.53.3. 变化

下面你会发现在PostgreSQL 8.4和之前主要发布之间变化的详细说明。

E.53.3.1. 性能

- 完善优化统计计算(Jan Urbanski, Tom)

  尤其是，估计全文搜索操作符被大幅度提高。

- 允许 `SELECT DISTINCT` 和 `UNION` / `INTERSECT` / `EXCEPT` 可以使用哈希(Tom)

  这意味着这些类型的查询不再自动生成有序输出。

- 创建半连接和反连接明确概念(Tom)

这项工作正规化 `IN(SELECT ...)` 子句之前的点对点处理， 并延伸到 `EXISTS` 和 `NOT EXISTS` 子句中。 它可能导致 `EXISTS` 和 `NOT EXISTS` 查询更好地规划。 一般情况下，逻辑上相同的 `IN` 和 `EXISTS` 子句有类似性能， 而此前的 `IN` 获胜。

- 提高外部链接下子选择优化(Tom)

  以前，一个子选择或视图不能被很好的优化， 如果它出现在外连接可空端并且在结果列中包含非严格表达式（比如，常量）。

- 通过使用Boyer-Moore-Horspool搜索提高 `text_position()` 和相关函数的性能(David Rowley)

  这对长期搜索模式特别有帮助。

- 当需要的时候，通过写文件降低写入统计收集文件I/O负载(Martin Pihlak)

- 提高批量插入性能(Robert Haas, Simon)

- 从 `10` 到 `100` 增加 `default_statistics_target` 缺省值(Greg Sabino Mullane,Tom)

  最大值从 `1000` 增加到 `10000` 。

- 在涉及继承或者 `UNION ALL` 的查询中缺省执行 `constraint_exclusion` 检查(Tom)

  新的 `constraint_exclusion` 设置， `分区` 被添加以指定该操作。

- 允许I/O预读取位图索引扫描(Greg Stark)

  通过 `effective_io_concurrency` 控制预读数量。 只有内核有 `posix_fadvise()` 支持时该功能可用。

- 内置 `FROM` 子句中简单集合返回SQL函数(Richard Rowell)

- 通过为连接密钥值提供一种特殊情况来优化多批次哈希连接性能， 这种情况在外部关系中特别常见(Bryce Cutt, Ramon Lawrence)

- 通过抑制"physical tlist"优化减少多批次哈希连接中临时数据量 (Michael Henderson, Ramon Lawrence)

- 避免 `CREATE INDEX CONCURRENTLY` 中等待空闲事务会话(Simon)

- 改进共享缓存无效性能(Tom)

E.53.3.2. 服务器

E.53.3.2.1. 设置

- 转换许多 `postgresql.conf` 设置以枚举值，使得 `pg_settings` 可以显示有效值(Magnus)

- 添加 `cursor_tuple_fraction` 参数以控制规划器假设被抓取的游标行(Robert Hell)
- 允许 `postgresql.conf` 中自定义变量类下的下划线(Tom)

E.53.3.2.2. 身份验证和安全

- 删除支持(不安全) `crypt` 认证方法(Magnus)

  这有效地淘汰了先前PostgreSQL7.2客户端库， 因为不再有他们可以使用的任何非明文口令方法。

- 支持 `pg_ident.conf` 中正则表达式(Magnus)
- 允许无需重启postmaster改变Kerberos/GSSAPI参数(Magnus)
- 支持服务器证书文件中SSL证书链(Andrew Gierth)

  包括完整证书链使客户能够验证证书， 而不需要存在于本地存储中的所有中间CA证书，这是商业CA的常见情况。

- 报告连接 `MD5` 认证和 `db_user_namespace` 启动的相应错误信息(Bruce)

E.53.3.2.3. `pg_hba.conf`

- 改变所有身份验证选项使用 `name=value` 语法(Magnus)

  这使得不兼容变化到 `ldap`，`pam` 和 `ident` 身份验证方法。 所有使用这些方法的 `pg_hba.conf` 项需要使用新的格式被重写。

- 删除 `ident sameuser` 选项， 如果没有指定usermap，而不是使得该操作缺省(Magnus)
- 允许为所有外部身份验证方法的usermap参数(Magnus)

  之前usermap只支持 `ident` 认证。

- 添加 `clientcert` 选项以控制客户端证书请求(Magnus)

  以前这是通过服务器的数据目录中根证书文件被控制。

- 添加 `cert` 认证方法以允许*user*认证通过SSL证书(Magnus)

  此前SSL证书只能验证客户端可以使用一个证书， 而不是验证用户。

- 允许在 `pg_hba.conf` 中指定 `krb5`，`gssapi` 和 `sspi` 字段和 `krb5` 主机设置(Magnus)

  这些覆盖了 `postgresql.conf` 中设置。

- 为 `krb5`， `gssapi` 和 `sspi` 方法添加 `include_realm` 参数(Magnus)

  这允许来自不同领域的相同用户名被认证为使用用户映射的不同数据库用户。

- 当它被加载的时候，完整解析 `pg_hba.conf` ，以致于 可以立即报告错误(Magnus)

  此前，文件中的大多数错误直到客户端试图连接时才会被检测到，所以一个错误的文件可能导致系统 无法使用。随着新的操作，如果在重载时检测到错误，那么坏文件被拒绝，并且postmaster继续使用其旧副本。

- 显示 `pg_hba.conf` 中所有解析错误 而不是在第一个之后终止(Selena Deckelmann)

- 支持Solaris上Unix域套接字上的 `ident` 认证(Garick Hamlin)

E.53.3.2.4. 连续归档

- 提供一个选项到 `pg_start_backup()` 以强制尽快完成的隐含检查点(Tom)

  缺省操作避免过多I/O消耗，但是如果没有并发查询活动，这是毫无意义的。

- 让 `pg_stop_backup()` 等待修改的WAL文件被归档(Simon)

  这保证备份在 `pg_stop_backup()` 完成时是有效的。

- 当启用归档时，在关闭时交替最后WAL段以致于所有事务可以立即被归档(Guillaume Smet, Heikki)

- 当连续归档基础备份在进展中时，延迟"smart"关机(Laurenz Albe)

- 如果请求"fast"关机，那么取消连续归档基础备份(Laurenz Albe)

- 允许 `recovery.conf` 布尔变量采取同一范围字符串值作为 `postgresql.conf` 布尔值(Bruce)

E.53.3.2.5. 监控

- 当最后加载PostgreSQL配置文件的时候，添加 `pg_conf_load_time()` 以报告(George Gensure)

- 添加 `pg_terminate_backend()` 以安全终止后台（ `SIGTERM` 信号也如此）(Tom, Bruce)

  虽然它一直可能 `SIGTERM` 单一后端， 这个以前被认为是不支持的； 并且测试的情况下发现了一些被修复的错误。

- 添加跟踪用户定义函数的调用计数以及运行时间能力(Martin Pihlak)

  函数统计出现在新的系统视图， `pg_stat_user_functions` 中。 通过新的参数 `track_functions` 控制跟踪。

- 在 `pg_stat_activity` 中通过新的 `track_activity_query_size` 参数 允许声明最大查询字符串大小(Thomas Lee)

- 增大最大线长度发送到syslog，希望优化性能(Tom)

- 添加只读配置变量 `segment_size` , `wal_block_size` 和 `wal_segment_size` (Bernd Helmle)

- 当报告死锁时，报告死锁服务器日志中涉及的所有查询文本(Itagaki Takahiro)
- 添加 `pg_stat_get_activity(pid)` 函数以返回关于指定进程id的信息(Magnus)
- 允许通过 `stats_temp_directory` 指定服务器的统计文件的位置(Magnus)

  这允许统计文件被放置在RAM固有目录中以减少I/O需求。 在启动/关闭时，文件被拷贝到它的传统位置(`$PGDATA/global/`)下， 所以重新启动时被保留。

E.53.3.3. 查询

- 添加支持 `WINDOW` 函数(Hitoshi Harada)
- 添加支持 `WITH` 子句(CTEs)，包含 `WITH RECURSIVE` (Yoshiyuki Asaba, Tatsuo Ishii, Tom)
- 添加 `TABLE` 命令(Peter)

  `TABLE tablename` 是 `SELECT * FROM tablename` 的一个SQL标准简写。

- 当声明 `SELECT` (或者 `RETURNING`)列输出标签的时候，允许 `AS` 为可选的(Hiroshi Saito)

  该操作只要列标签没有任何PostgreSQL关键字，否则仍然需要 `AS` 。

- 支持在 `SELECT` 结果列表中设置返回函数， 即使对于通过tuplestore返回其结果的函数(Tom)

  特别是，这意味着用PL/pgSQL和其它PL/pgSQL语言书写的函数可以这样被调用。

- 在聚集和分组查询输出中支持设置返回函数(Tom)
- 允许 `SELECT FOR UPDATE` / `SHARE` 在继承树上进行操作(Tom)
- 为SQL/MED添加基础设施(Martin Pihlak, Peter)

  然而没有远程或外部SQL/MED功能， 但这种变化提供了一个标准化的，面向未来系统，用于管理类似模块 `dblink` 和 `plproxy` 连接信息。

- 当引用的模式，函数，操作符或者操作符类被修改的时候，使缓存计划无效(Martin Pihlak, Tom)

  这提高了系统对反应传输中DDL变化的能力。

- 允许复合类型比较以及匿名复合类型的数组(Tom)

  这允许构造比如 `row(1, 1.1) = any (array[row(7, 7.7), row(1, 1.0)])` 。 在递归查询中特别有用。

- 添加使用代码点支持Unicode字符串文本和标识符规范， 比如 `U&'d\0061t\+000061'` (Peter)

- 拒绝字符串文本中 `\000` 和 `COPY` 数据(Tom)

  之前，这被接受但是有终止字符串内容的影响。

- 提高解析器报告错误位置的能力(Tom)

  为许多语义错误报道一个错误位置， 比如不匹配的数据类型，这在以前不可能被本地化。

E.53.3.3.1. `TRUNCATE`

- 支持语句级别 `ON TRUNCATE` 触发器(Simon)
- 为了 `TRUNCATE TABLE` 添加 `RESTART` / `CONTINUE IDENTITY` 选项(Zoltan Boszormenyi)

  序列的起始值可以通过 `ALTER SEQUENCE START WITH` 被改变。

- 允许 `TRUNCATE tab1, tab1` 成功(Bruce)
- 添加单独 `TRUNCATE` 许可(Robert Haas)

E.53.3.3.2. `EXPLAIN`

- 采用 `EXPLAIN VERBOSE` 显示每个规划节点的输出列(Tom)

  此前 `EXPLAIN VERBOSE` 输出查询规划的内部表示。 （该操作通过 `debug_print_plan` 是可用的。）

- 采用 `EXPLAIN` 标识子计划以及单一标签的初始计划(Tom)
- 为了 `debug_print_plan` 采用 `EXPLAIN` (Tom)
- 允许在 `CREATE TABLE AS` 上进行 `EXPLAIN` (Peter)

E.53.3.3.3. `LIMIT` / `OFFSET`

- 允许 `LIMIT` 和 `OFFSET` 中子选择(Tom)
- 为了 `LIMIT` / `OFFSET` 性能添加SQL标准语法(Peter)

  也就是说， `OFFSET num {ROW|ROWS} FETCH {FIRST|NEXT} [num] {ROW|ROWS} ONLY` 。

E.53.3.4. 对象操作

- 添加列级权限支持(Stephen Frost, KaiGai Kohei)
- 重构多目标 `DROP` 操作以减少 `CASCADE` 的需要(Alex Hunsaker)

  比如，如果表 `B` 在表 `A` 上有依赖物，那么 命令 `DROP TABLE A, B` 不再需要 `CASCADE` 选项。

- 通过确保在开始删除对象依赖之前采取锁定，修复各种并发 `DROP` 命令问题(Tom)
- 改善 `DROP` 命令中依赖性报告(Tom)
- 添加 `WITH [NO] DATA` 子句给 `CREATE TABLE AS` ，按照SQL标准(Peter, Tom)
- 添加支持用户自定义I/O转换模型(Heikki)
- 允许 `CREATE AGGREGATE` 使用 `internal` 转换数据类型(Tom)
- 添加 `LIKE` 子句到 `CREATE TYPE` (Tom)

  这简化了使用作为已存在类型同一内部表示形式的数据类型的创建。
- 允许类型类别指定以及用户定义的基础类型"首选"状态(Tom)

  这允许控制用户定义类型的强制操作。
- 允许 `CREATE OR REPLACE VIEW` 添加列到视图结尾(Robert Haas)

E.53.3.4.1. `ALTER`

- 添加 `ALTER TYPE RENAME` (Petr Jelinek)
- 添加 `ALTER SEQUENCE ... RESTART` （无参数）以重置序列到其初始值(Zoltan Boszormenyi)
- 修改 `ALTER TABLE` 语法以允许表，索引，序列以及视图合理组合(Tom)

  这种变化允许下面新语法：

  - `ALTER SEQUENCE OWNER TO`
  - `ALTER VIEW ALTER COLUMN SET/DROP DEFAULT`
  - `ALTER VIEW OWNER TO`
  - `ALTER VIEW SET SCHEMA`

  这里没有实际的新功能， 但之前你不得不表明 `ALTER TABLE` 可以做这些事情，使其混淆。
- 添加支持语法 `ALTER TABLE ... ALTER COLUMN... SET DATA TYPE` (Peter)

  这是已经支持的功能性SQL标准语法。
- 采用 `ALTER TABLE SET WITHOUT OIDS` 重写表以物理删除 `OID` 值(Tom)

  同时添加 `ALTER TABLE SET WITH OIDS` 重写表添加 `OID` 。

E.53.3.4.2. 数据库操作

- 当未提交预备事务的时候，改善`CREATE` / `DROP` / `RENAME DATABASE`错误报告(Tom)
- 每个数据库设置采用`LC_COLLATE`和`LC_CTYPE`(Radek Strnad, Heikki)

  整理类似编码，总是配置每个数据库。

- 完善检查数据库编码，整理(`LC_COLLATE`), 以及字符类(`LC_CTYPE`)匹配(Heikki, Tom)

  特别注意，当从`template0`拷贝的时候，一个新的数据库编码和语言环境设置被改变。这避免了可能拷贝的数据不匹配该设置。

- 添加`ALTER DATABASE SET TABLESPACE`移动数据库到一个新的表空间(Guillaume Lelarge, Bernd Helmle)

E.53.3.5. 实用操作

- 添加`VERBOSE`选项到`CLUSTER`命令和clusterdb中(Jim Cox)
- 为记录待定的触发器事件减少内存需求(Tom)

E.53.3.5.1. 索引

- 大幅提高建立并且访问哈希索引速度(Tom Raney, Shreya Bhargava)

  这使得散列索引有时比btree索引的速度更快。 然而，哈希索引仍然没有碰撞安全。

- 使得哈希索引只存储哈希码，而不是索引列的完整值(Xiao Meng)

  这大大减少了为长索引值的散列索引的大小，从而提高了性能。

- 实现GIN索引的快速更新选项(Teodor, Oleg)

  此选项大大提高了在搜索速度中小的不利后果上的更新速度。

- `xxx_pattern_ops`索引可以用于简单平等比较，而不仅仅为`LIKE`(Tom)

E.53.3.5.2. 全文索引

- 当在全文索引中执行GIN加权查找时，删除该需求而使用`@@@`(Tom, Teodor)

  可以使用正常的`@@`文本搜索操作符。

- 为`@@`文本搜索操作添加优化选择函数(Jan Urbanski)
- 允许全文搜索中前缀匹配(Teodor Sigaev,Oleg Bartunov)
- 支持多列GIN索引(Teodor Sigaev)
- 改善支持Nepali语言和Devanagari字母(Teodor)

E.53.3.5.3. `VACUUM`

- 在单独每个关系"fork"文件中跟踪自由空间(Heikki)

  通过 `VACUUM` 发现的自由空间在 `*_fsm` 文件中被记录， 而不是在一个固定大小的共享存储器区域中。 `max_fsm_pages` 和 `max_fsm_relations` 设置已被移除， 从而大大简化了自由空间管理。

- 添加可见视图以跟踪不需要清理页面(Heikki)

  这允许 `VACUUM` 以避免扫描所有表， 只有表中的一部分需要清理的时候。可见视图被存储在每个关系"fork"文件。

- 添加 `vacuum_freeze_table_age` 参数以控制 当 `VACUUM` 忽略可见视图并且执行全表扫描以冻结元组。

- 更仔细追踪事务快照(Alvaro)

  这提高了 `VACUUM` 在长时间运行的事务中回收空间能力。

- 添加指定每个关系autovacuum以及 `CREATE TABLE` 中TOAST参数的能力(Alvaro, Euler Taveira de Oliveira)

  自动清理选项存储在系统表中。

- 添加 `--freeze` 选项到vacuumdb(Bruce)

E.53.3.6. 数据类型

- 为文本搜索同义词词典添加 `CaseSensitive` 选项(Simon)
- 完善 `NUMERIC` 划分精度(Tom)
- 为 `int2` 与 `int8` 添加基础算术操作符(Tom)

  这消除了在某些情况中显式投射需求。

- 允许 `UUID` 输入在每四位后面接受可选的连字符(Robert Haas)
- 允许 `on` / `off` 作为布尔数据类型的输入(Itagaki Takahiro)
- 为类型 `numeric` 在输入字符串中 `NaN` 周围允许空格(Sam Mason)

E.53.3.6.1. 时间数据类型

- 拒绝年 `0 BC` 以及 `000` 和 `0000` (Tom)

  以前这些被解释为 `1 BC` 。（注意：年 `0` 和 `00` 仍然假定为2000。）

- 在已知时区缩写缺省列表中包含 `SGT` (Singapore时间)

- 支持 `infinity` 和 `-infinity` 作为类型 `date` 的值(Tom)
- 使得 `interval` 文本解析更加遵循标准(Tom, Ron Mayer)

  比如， `INTERVAL '1' YEAR` 目前执行应该做的内容。

- 在 `second` 关键字之后，按照SQL标准允许指定 `interval` 分数秒精度(Tom)

  之前在关键字 `interval` 之后指定精度。（为了向后兼容，这种语法仍 支持，但不建议使用。）数据类型定义现在使用标准格式被输出。

- 支持 ISO 8601 `interval` 语法(Ron Mayer, Kevin Grittner)

  比如，现在支持 `INTERVAL 'P1Y2M3DT4H5M6.7S'` 。

- 添加 `IntervalStyle` 参数以控制 `interval` 值如何输出(Ron Mayer)

  有效值是： `postgres` , `postgres_verbose` , `sql_standard` , `iso_8601` 。 只有当一些字段有正/负名称的时候，该设置也控制负 `interval` 输入的处理。

- 在 `timestamp` 和 `interval` 输入中优化小数秒处理的一致性(Ron Mayer)

E.53.3.6.2. 数组

- 优化适用于 `ARRAY[]` 结构映射处理， 比如 `ARRAY[...]::integer[]` (Brendan Jurd)

  之前PostgreSQL试图为 `ARRAY[]` 结构而不引用随后计算确定一个数据类型。 这在许多情况下可能失败，特别是当 `ARRAY[]` 构造是空的或仅包含不明确项如 `NULL` 的时候。 现在协商该计算以确定该数组元素类型。

- 使得SQL语法 `ARRAY` 尺寸可选以匹配SQL标准(Peter)
- 添加 `array_ndims()` 以返回数组维数(Robert Haas)
- 添加 `array_length()` 以返回指定维数数组长度(Jim Nasby, Robert Haas, Peter Eisentraut)
- 添加聚集函数 `array_agg()` 作为一个数组返回所有聚集值(Robert Haas,Jeff Davis, Peter)
- 添加 `unnest()` 将一个数组转换为各行值(Tom)

  这是 `array_agg()` 的反面。

- 添加 `array_fill()` 以创建初始化值数组(Pavel Stehule)
- 添加 `generate_subscripts()` 以简化生成数组下标范围(Pavel Stehule)

E.53.3.6.3. 宽值存储(TOAST)

- 认为TOAST压缩值和32字节一样短（之前256字节）(Greg Stark)

- 使用TOAST压缩之前要求最低25%的空间节省，（之前20%为小值并且任何存储为大值）(Greg)
- 优化具有较大和较小toastable字段混合行的TOAST启发式， 使我们更愿意使得较大值超出范围并且不压缩较小值(Greg, Tom)

E.53.3.7. 函数

- `setseed()` 文件允许从 `-1` 到 `1` 的值（不只是 `0` 到 `1` ）， 并且执行有效范围(Kris Jurka)
- 添加服务器端函数 `lo_import(filename, oid)` (Tatsuo)
- 添加 `quote_nullable()` 其操作类似于 `quote_literal()` 但为空参数返回字符串 `NULL` (Brendan Jurd)
- 优化全文本搜索 `headline()` 函数以允许提取文本若干片段(Sushant Sinha)
- 添加 `suppress_redundant_updates_trigger()` 触发器函数以避免 无数据变化更新开销(Andrew)
- 添加 `div(numeric, numeric)` 执行 `numeric` 除法而没有四舍五入(Tom)
- 添加 `generate_series()` 的 `timestamp` 和 `timestamptz` 版本(Hitoshi Harada)

E.53.3.7.1. 对象信息函数

- 通过需要知道当前正在运行的查询的函数实现 `current_query()` 的使用(Tomas Doran)
- 添加 `pg_get_keywords()` 返回解析关键字列表(Dave Page)
- 添加 `pg_get_functiondef()` 查看函数的定义(Abhijit Menon-Sen)
- 当解析不包含变量的表达式时，允许 `pg_get_expr()` 第二个参数为零(Tom)
- 修改 `pg_relation_size()` 以使用 `regclass` (Heikki)

  `pg_relation_size(data_type_name)` 不再起作用。
- 添加 `boot_val` 和 `reset_val` 列到 `pg_settings` 输出(Greg Smith)
- 为配置文件中变量设置添加源文件名和行号列到 `pg_settings` 输出(Magnus, Alvaro)

  出于安全原因，这些列只对超级用户可见。
- 添加支持 `CURRENT_CATALOG` , `CURRENT_SCHEMA` , `SET CATALOG` , `SET SCHEMA` (Peter)

  为存在特性提供SQL标准语法。
- 添加 `pg_typeof()` 返回任何值的数据类型(Brendan Jurd)

- 采取 `version()` 返回有关服务器是否是32或者64位二进制信息(Bruce)
- 修复信息模式列 `is_insertable_into` 和 `is_updatable` 操作是一致的(Peter)
- 改善信息模式 `datetime_precision` 列的操作(Peter)

  这些列为 `date` 列显示为零并且 `time` ， `timestamp` 以及 没有声明精度的 `interval` 显示为6（默认精度），而不如以前一样显示为空。

- 转换剩余内置集合返回函数使用 `OUT` 参数(Jaime Casanova)

  这使得它可以调用没有指定列表的函数： `pg_show_all_settings()` ， `pg_lock_status()` ， `pg_prepared_xact()` ， `pg_prepared_statement()` ， `pg_cursor()`

- 采用 `pg_*_is_visible()` 和 `has_*_privilege()` 为无效OID返回 `NULL` ， 而不是抛出错误(Tom)
- 扩展 `has_*_privilege()` 函数以允许查询一个调用中多特权的OR(Stephen Frost, Tom)
- 添加 `has_column_privilege()` 和 `has_any_column_privilege()` 函数(Stephen Frost, Tom)

E.53.3.7.2. 函数创建

- 支持可变参数的函数（带有参数可变数量的函数）(Pavel Stehule)

  只有尾随参数是可选的，它们都必须是相同的数据类型。

- 支持函数参数的缺省值(Pavel Stehule)
- 添加 `CREATE FUNCTION ... RETURNS TABLE` 子句(Pavel Stehule)
- 允许SQL语言函数返回 `INSERT` / `UPDATE` / `DELETE` `RETURNING` 子句的输出(Tom)

E.53.3.7.3. PL/pgSQL服务器端语言

- 为了更容易地将数据值插入到动态查询字符串中支持 `EXECUTE USING` (Pavel Stehule)
- 允许循环使用 `FOR` 循环游标结果(Pavel Stehule)
- 支持 `RETURN QUERY EXECUTE` (Pavel Stehule)
- 优化 `RAISE` 命令(Pavel Stehule)
  - 支持 `DETAIL` 和 `HINT` 字段
  - 支持 `SQLSTATE` 错误代码规范
  - 支持异常名字参数
  - 允许异常块中无参数 `RAISE` 重新抛出当前错误

- 允许 `EXCEPTION` 列表中 `SQLSTATE` 代码规范(Pavel Stehule)

  这对于处理自定义 `SQLSTATE` 代码非常有用。

- 支持 `CASE` 语句(Pavel Stehule)

- 使用 `RETURN QUERY` 设置特殊的 `FOUND` 和 `GET DIAGNOSTICS` `ROW_COUNT` 变量(Pavel Stehule)

- 使用 `FETCH` 和 `MOVE` 设置 `GET DIAGNOSTICS` `ROW_COUNT` 变量(Andrew Gierth)

- 使用没有标签的 `EXIT` 总是退出最内层循环(Tom)

  之前，如果有一个 `BEGIN` 块比任何环路更加紧密的嵌套， 将退出该块。新的操作匹配Oracle(TM)并且也正如通过自身文档所讲述的。

- 使得字符串文本处理和嵌套块注释匹配主SQL解析器的处理(Tom)

  特别是，在 `RAISE` 中格式字符串执行与任何其他字符串文本一样的操作， 包含从属于 `standard_conforming_strings` 。当 `standard_conforming_strings` 为on的时候， 这种变化也修复了有效命令失败的情况。

- 当在不同异常块嵌套深度中调用同一函数的时候，避免内存泄露(Tom)

E.53.3.8. 客户端应用

- 修复 `pg_ctl restart` 保存命令行参数(Bruce)

- 添加 `-w` / `--no-password` 选项防止在具有 `-W` / `--password` 选项实用程序中的密码提示(Peter)

- 删除createdb, createuser, dropdb, dropuser中的 `-q` (默默的)选项(Peter)

  这些选项不起作用因为PostgreSQL 8.3。

E.53.3.8.1. psql

- 删除详细的启动标志，目前建议 `help` (Joshua Drake)

- 使用 `help` 显示常见的反斜杠命令(Greg Sabino Mullane)

- 添加 `\pset格式包` 模式来包装输出到屏幕宽度， 或者文件/管道输出，如果设置 `\pset列` (Bryce Nesbitt)

- 允许 `\pset` 中布尔值的所有支持的拼写，而不仅仅 `on` 和 `off` (Bruce)

  之前，除了"off"外的任何字符串默默地认为 `true` 。 psql抱怨无法识别的拼写（但是仍然认为 `true` ）。

- 使用宽输出分页程序(Bruce)
- 在一个字母反斜杠命令和它的第一个参数之间需要一个空格(Bernd Helmle)

  这消除了有歧义的历史渊源。

- 改善标签完整支持模式修饰以及带引号标识符(Greg Sabino Mullane)
- 为 `\timing` 添加可选的 `on` / `off` 参数(David Fetter)
- 显示多行上的访问控制权(Brendan Jurd, Andreas Scherbaum)
- 使用 `\l` 显示数据库访问权限(Andrew Gilligan)
- 使用 `\l+` 显示数据库大小，如果权限允许(Andrew Gilligan)
- 添加 `\ef` 命令编辑函数定义(Abhijit Menon-Sen)

E.53.3.8.2. psql \d* 命令

- 使得没有模式参数的 `\d*` 命令显示系统对象， 只有当指定 `S` 修饰语的时候(Greg Sabino Mullane, Bruce)

  前者操作与 `\d` 不同形式不一致，并且在大多数情况下它没有提供简单方法查看 用户对象。

- 完善与旧的PostgreSQL服务器版本一起执行（追溯到7.4）的 `\d*` 命令， 而不仅仅是当前服务器版本(Guillaume Lelarge)
- 使用 `\d` 显示引用已选择表的外键约束(Kenneth D'Souza)
- 在序列上使用 `\d` 显示列值(Euler Taveira de Oliveira)
- 添加列存储类型和其他关系选项到 `\d+` 显示中(Gregory Stark, Euler Taveira de Oliveira)
- 显示 `\dt+` 输出中关系大小(Dickson S. Guedes)
- 显示 `\dT+` 中 `enum` 类型可能值(David Fetter)
- 允许 `\dC` 接受通配符模式，这可以匹配涉及到投射的数据类型(Tom)
- 添加函数类型列到 `\df` 的输出，并且添加选项只列出函数所选类型(David Fetter)
- `\df` 不隐藏使用或者返回类型 `cstring` 的函数(Tom)

  此前，这种函数被隐藏，因为他们大多是数据类型I/O函数，这被认为是没有意义的。 默认情况下有关隐藏系统函数的新策略使得该缺点不必要的。

E.53.3.8.3. pg_dump

- 添加 `--no-tablespaces` 选项到pg_dump/pg_dumpall/pg_restore 这样转储可以恢复到具有非匹配表空间层次的集群中(Gavin Roy)
- 从pg_dump和pg_dumpall中删除 `-d` 和 `-D` 选项(Tom)

  这些选项不应该过于频繁地与在其它PostgreSQL客户端应用程序中选择数据库名称的选项混淆。 该功能仍然可用， 但你现在必须拼写出长选项名称 `--inserts` 或者 `--column-inserts` 。

- 从pg_dump和pg_dumpall中删除 `-i` / `--ignore-version` 选项(Tom)

  使用此选项不会抛出一个错误，但它没有效果。 该选项被移除，因为版本检查对安全性是必要的。

- 在转储和恢复中禁用 `statement_timeout` (Joshua Drake)
- 添加pg_dump/pg_dumpall选项 `--lock-wait-timeout` (David Gould)

  如果在指定时间内无法获取共享锁，那么转储失败。

- 重新排序pg_dump `--data-only` 输出 用来转储在引用表之前通过外键参考的表(Tom)

  当外键存在的时候，允许数据加载。如果循环引用采用一个不可能的安全排序， 发出 `NOTICE` 。

- 允许pg_dump, pg_dumpall和pg_restore使用指定用户(Benedek László)
- 允许pg_restore使用多个并发连接执行恢复(Andrew)

  通过选项 `--jobs` 来控制连接数。这仅仅支持自定义格式归档。

E.53.3.9. 编程工具

E.53.3.9.1. libpq

- 当通过新函数 `lo_import_with_oid()` 导入大对象的时候，允许指定 `OID` (Tatsuo)
- 添加"events"支持(Andrew Chernow, Merlin Moncure)

  这添加了注册回调能力来管理联系 `PGconn` 和 `PGresult` 对象的私有数据。

- 优化错误处理以允许多个错误信息的返回作为多行错误报告(Magnus)
- 采用 `PQexecParams()` 和相关函数为空查询返回 `PGRES_EMPTY_QUERY` (Tom)

  之前返回 `PGRES_COMMAND_OK` 。

- 记录Windows上如何避免 `WSACleanup()` 的开销(Andrew Chernow)
- 不依赖于Kerberos来决定缺省数据库用户名(Magnus)

之前，libpq的Kerberos编译将使用从任何可用的Kerberos标签作为缺省数据库用户名的主体名称， 即使连接没有使用Kerberos认证。 这被认为是不一致而且混乱的。 有或没有Kerberos同样的方式决定缺省用户名。 然而，当使用Kerberos身份验证的时候，需要注意的是数据库用户名必须匹配该标签。

E.53.3.9.2. libpq SSL (安全套接层)支持

- 修复SSL连接的证书验证(Magnus)

  libpq现在支持验证两种证书 和SSL连接时服务器的名称。 如果根证书不可用于验证，那么SSL连接失败。 `sslmode` 参数用于启用证书验证，并设置检查水平。 默认值还没有做任何验证，允许连接到不需要客户端根证书的SSL启用服务器中。 ????????

- 支持通配符服务器证书(Magnus)

  如果证书CN以 `*` 开头，它将被视为一个通配符，当匹配hostname的时候， 允许多个服务器同一证书的使用。

- 允许文件位置指定客户端证书(Mark Woodward, Alvaro, Magnus)

- 增加 `PQinitOpenSSL` 函数允许更大控制OpenSSL/libcrypto初始化(Andrew Chernow)

- 当没有数据库连接保持打开的时候， 使用libpq注销它的OpenSSL回调(Bruce, Magnus, Russell Smith)

  卸载libpq库的应用程序是必须的，否则无效的OpenSSL回调将保持不变。

E.53.3.9.3. ecpg

- 添加信息本地化支持(Euler Taveira de Oliveira)

- 从服务器解析器自动生成ecpg解析器(Michael)

  之前ecpg解析器手动维护。

E.53.3.9.4. 服务器编程接口(SPI)

- 添加超出行参数的单一使用规划(Tom)

- 添加新的 `SPI_OK_REWRITTEN` 返回 `SPI_execute()` 代码(Heikki)

  当命令被重写为命令的另一种类型的时候使用它。

- 从 `executor/spi.h` 中删除不必要内含物(Tom)

  如果它们依赖 `spi.h` 包含它们需要的东西， SPI使用模块可能需要添加一些 `#include` 行。

E.53.3.10. 编译选项

- 使用Autoconf 2.61更新编译系统(Peter)
- 源码编译需要GNU bison(Peter)

  这个有效的使用了许多年，但是现在没有基础设施声明支持其它解析工具。
- 添加pg_config `--htmldir` 选项(Peter)
- 通过内部服务器传递 `float4` 值(Zoltan Boszormenyi)

  添加configure选项 `--disable-float4-byval` 以使用旧操作。 使用旧式（版本0）的外部C函数调用约定 并通过这个改变打破传递或者返回 `float4` 值， 如果你具有这样的函数，并且不想更新它们， 所以你可能需要configure选项。
- 通过64位平台上内部服务器传递 `float8` , `int8` 和相关数据类型值(Zoltan Boszormenyi)

  添加configure选项 `--disable-float8-byval` 用来使用旧的操作。如上所述， 该变化可能打破旧形式外部C函数。
- 添加配置选项 `--with-segsize` , `--with-blocksize` , `--with-wal-blocksize` , `--with-wal-segsize` (Zdenek Kotala, Tom)

  这简化了编译时控制之前只能通过编辑 `pg_config_manual.h` 来改变的几个常数。
- 允许在Solaris 2.5上线程编译(Bruce)
- 在Solaris上使用系统的 `getopt_long()` (Zdenek Kotala, Tom)

  这使得选项处理与Solaris用户期望的更加一致。
- 添加Linux上Sun Studio编译器支持(Julius Stroffek)
- 追加主版本号到后端gettext域，而 `soname` 主版本号 到库的gettext域(Peter)

  这简化了多个版本并行安装。
- 允许支持gcov代码覆盖测试(Michelle Caisse)
- 允许在Mingw和Cygwin上树外编译(Richard Evans)
- 修复作为交叉编译源平台Mingw的使用(Peter)

E.53.3.11. 源代码

- 支持64位时区数据文件(Heikki)

  这添加支持超出2038年夏令时(DST)计算。

- 不赞成平台的 `time_t` 数据类型的使用(Tom)

  有些平台已经迁移到64位 `time_t` ， 有些不能，Windows无法下定决心它在做什么。 定义 `pg_time_t` 和 `time_t` 具有相同的含义， 但始终是64位（除非该平台不具有64位整数类型）， 并且使用所有模块API中的类型和磁盘上的数据格式。

- 当交叉编译的时候，修复时区数据库处理中错误(Richard Evans)
- 在一个步骤中连接后端对象文件，而不是一个阶段中(Peter)
- 完善gettext支持以便复数的更好转化(Peter)
- 添加PL语言的信息翻译支持(Alvaro, Peter)
- 添加更多DTrace探测(Robert Lor)
- 启用Mac OS X Leopard以及其它非Solaris平台上的DTrace支持(Robert Lor)
- 简化并且标准化C字符串和 `text` 数据之间转换，出于此目的提供普通函数(Brendan Jurd, Tom)
- 清理 `include/catalog/` 头文件以致于前端程序包含它们而不包含 `postgres.h` (Zdenek Kotala)
- 采取 `name` 字符对齐，并且抑制索引中 `name` 项的零填充(Tom)
- 如果动态加载代码执行 `exit()` 更好恢复(Tom)
- 添加连接让插件监控执行者(Itagaki Takahiro)
- 添加连接允许规划器的统计查找操作被覆盖(Simon Riggs)
- 为自定义共享内存需求添加 `shmem_startup_hook()` (Tom)
- 使用 `amgetbitmap` 替换索引访问方法 `amgetmulti` 切入点， 并且为 `amgettuple` 扩展API可以支持操作符丢失的运行时计算(Heikki, Tom, Teodor)

  为GIN和GiST opclass `consistent` 函数的API已经被扩展。

- 添加支持GIN索引中局部匹配搜索(Teodor Sigaev, Oleg Bartunov)
- 使用布尔 `relhastriggers` 替换 `pg_class` 列 `reltriggers` (Simon)

  同时删除未使用的 `pg_class` 列 `relukeys` ， `relfkeys` 和 `relrefs` 。

- 添加 `relistemp` 列到 `pg_class` 解除临时表识别(Tom)
- 将平台FAQ放到主文档中(Peter)
- 防止解析器输入文件编译冲突(Peter)

- 添加 `KOI8U` (Ukrainian)编码(Peter)
- 添加Japanese信息转化(Japan PostgreSQL用户组)

  这用于作为单一项目被维护。

- 当在MSVC编译系统上设置 `LC_MESSAGES` 的时候， 修复该问题(Hiroshi Inoue, Hiroshi Saito, Magnus)

E.53.3.12. Contrib

- 添加 `contrib/auto_explain` 在查询超过指定时间上自动运行 `EXPLAIN` (Itagaki Takahiro, Tom)
- 添加 `contrib/btree_gin` 允许GIN索引处理更多数据类型(Oleg, Teodor)
- 添加 `contrib/citext` 提供不区分大小写，多字节文本数据类型(David Wheeler)
- 为语句执行统计的服务器范围追踪添加 `contrib/pg_stat_statements` (Itagaki Takahiro)
- 添加时间和查询模式选项到 `contrib/pgbench` (Itagaki Takahiro)
- `contrib/pgbench` 使用表名 `pgbench_accounts` , `pgbench_branches` , `pgbench_history` 和 `pgbench_tellers` , 而不仅仅 `accounts` , `branches` , `history` 和 `tellers` (Tom)

  通过运行pgbench以降低意外破坏真实数据的风险。

- 修复 `contrib/pgstattuple` 处理表和超过20亿页面的索引(Tatsuhito Kasahara)
- 在 `contrib/fuzzystrmatch` 中，添加Levenshtein字符串距离函数版本允许 用户声明插入， 删除和替换成本(Volkan Yazici)
- 采用 `contrib/ltree` 支持多字节编码(laser)
- 启用 `contrib/dblink` 使用存储在SQL/MED目录中的连接信息(Joe Conway)
- 完善来自远程服务器错误的 `contrib/dblink` 的报告(Joe Conway)
- 使用 `contrib/dblink` 设置 `client_encoding` 以 匹配本地数据库的编码(Joe Conway)

  当与使用不同编码的远程数据库通信时，这可以避免编码问题。

- 确保 `contrib/dblink` 使用用户提供的密码， 并且不是意外地采取服务器的 `.pgpass` 文件(Joe Conway)

  有一个小的安全增强功能。

- 添加 `fsm_page_contents()` 到 `contrib/pageinspect` (Heikki)

- 修改 `get_raw_page()` 支持自由空间映射(`*_fsm`)文件。 同时更新 `contrib/pg_freespacemap` 。
- 添加支持多字节编码到 `contrib/pg_trgm` (Teodor)
- 改写 `contrib/intagg` 使用新的函数 `array_agg()` 和 `unnest()` (Tom)
- 故障转移之前使用 `contrib/pg_standby` 回收所有可用WAL(Fujii Masao, Simon, Heikki)

  为了使这项工作安全，您现在需要设置 `recovery.conf` 中新的 `recovery_end_command` 选项用来清理故障转移后的触发器文件。 pg_standby将不再删除触发器文件本身。

- `contrib/pg_standby` 的 `-l` 选项现在是一个空操作， 因为它使用符号链接不安全(Simon)

E.54. 发布8.3.23

> 发布日期: 2013-02-07

该发布包含来自8.3.22中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

预计这是8.3.X系列中最后PostgreSQL发布。 鼓励用户尽快更新到新版本分支。

E.54.1. 迁移到版本8.3.23

运行8.3.X不需要备份/恢复。

然而，如果从8.3.17更早版本更新，参阅8.3.17发布说明。

E.54.2. 变化

- 避免来自SQL `enum_recv` 的执行(Tom Lane)

  该函数被错误声明，使得简单SQL命令造成服务器崩溃。 原则上攻击者可能使用它检查服务器内存的内容。 非常感谢Sumit Soni(通过Secunia SVCRP)报告这个问题(CVE-2013-0255)

- 修复SQL语法以允许来自子SELECT结果的下标或者字段选择(Tom Lane)

- 当扫描 `pg_tablespace` 的时候，防止竞态条件(Stephen Frost, Tom Lane)

  如果有 `pg_tablespace` 项的并发更新， 那么 `CREATE DATABASE` 和 `DROP DATABASE` 可能误操作。

- 防止 `DROP OWNED` 尝试删除整个数据库或者表空间(Álvaro Herrera)

  出于安全考虑，这些对象的所有权被重新分配，而不是被删除。

- 当 `RowExpr` 或者 `XmlExpr` 被解析分析两次的时候， 避免误操作(Andres Freund, Tom Lane)

  该错误在上下文中是用户可见的，比如 `CREATE TABLE LIKE INCLUDING INDEXES` 。

- 提高在哈希大小计算中防御整数溢出(Jeff Davis)

- 确保非ASCII提示字符串被转换为Windows上正确代码页(Alexander Law, Noah Misch)

  这个错误影响psql以及一些其他的客户端程序。

- 当不连接到数据库的时候，修复psql的 `\?` 命令中可能崩溃(Meng Qingzhong)

- 修复libpq的 `PQprintTuples` 中一个字节缓冲区溢出(Xi Wang)

  这个以往函数不会被PostgreSQL自身使用，但是它可能 通过客户端代码被使用。

- 为提供的函数重新分配配置测试， 因此它不会通过libedit/libreadline虚假输出被愚弄(Christoph Berg)

- 确保随时间推移Windows编译数增加(Magnus Hagander)

- 当Windows交叉编译时，使得pgxs建立具有正确 `.exe` 后缀的可执行文件(Zoltan Boszormenyi)

- 添加新的时区缩写 `FET` (Tom Lane)

  这被用于一些东欧时区。

E.55. 发布8.3.22

> 发布日期: 2012-12-06

该发布包含来自8.3.21中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

PostgreSQL社区在2013年2月为8.3.X发布系列停止发布更新。 鼓励用户尽快更新到新版本分支。

E.55.1. 迁移到版本8.3.22

运行8.3.X不需要备份/恢复。

然而，如果从8.3.17更早版本更新，参阅8.3.17发布说明。

E.55.2. 变化

- 修复与 `CREATE INDEX CONCURRENTLY` 相关的多个错误(Andres Freund, Tom Lane)

  当改变索引 `pg_index` 行状态的时候，修复 `CREATE INDEX CONCURRENTLY` 以使用适当更新。 这可以防止竞争条件，它可以导致并发会话错过更新目标索引，从而造成损坏同时创建索引。

  此外，修复其他各种操作以确保他们忽视由失败 `CREATE INDEX CONCURRENTLY` 导致的无效索引。 最重要的是，这些是 `VACUUM` ，因为 在该表上采取纠正措施以修复或者删除无效索引之前自动清理很容易被发动。

- 当内存不足的时候，避免内部哈希表的损坏(Hitoshi Harada)

- 修复外部连接上非严格等价子句的规划(Tom Lane)

  规划器可以从子句等同于非严格结构中别的方面获取不正确限制， 比如 `WHERE COALESCE(foo, 0) = 0` 当 `foo` 来自外部链接的空侧端时。

- 完善规划器来自等价类排除约束能力(Tom Lane)

- 修复散列子查询中部分行匹配以正确处理交叉类型情况(Tom Lane)

  这影响了多列 `NOT IN` 子计划，比如 `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` 。 当比如 `b` 和 `y` 分别为 `int4` 和 `int8` 的时候。 这个错误导致了错误结果或者依赖于涉及到的特定数据类型的崩溃。

- 当为 `AFTER ROW UPDATE/DELETE` 触发器重新读取旧元组的时候， 获取缓冲锁(Andres Freund)

  在非常特殊的情况下，这种疏忽可能导致传递 不正确的数据来预检查外键执行触发器逻辑。 这可能会导致系统崩溃，或者关于是否触发触发器的一个不正确的决定。

- 修复 `REASSIGN OWNED` 以处理在表空间上的授权(Álvaro Herrera)

- 为视图上系统列忽略不正确 `pg_attribute` 项(Tom Lane)

  视图没有任何系统列。但是，当转换表到视图时，我们忘了删除这些项。 这在9.3及更高版本中被正确修复，但在以往的分支中我们需要防御现有错误转换视图。

- 修复规则输出以正确备份 `INSERT INTO` `_table_` DEFAULT VALUES(Tom Lane)

- 当在查询中有许多 `UNION` / `INTERSECT` / `EXCEPT` 子句的时候， 防止堆栈溢出(Tom Lane)

- 当最小可能整数值除以-1的时候，避免平台依赖错误(Xi Wang, Tom Lane)

- 修复日期解析中可能访问字符串结束端(Hitoshi Harada)

- 如果Unix域套接字路径名长度超过指定平台限制，那么产生可理解错误消息(Tom Lane, Andrew Dunstan)

  之前，这可能导致一些无用的东西，比如"域名解析中不可恢复故障"。

- 当发送复合列值到客户端的时候，修复内存泄露(Tom Lane)

- 采取pg_ctl使得读取 `postmaster.pid` 文件更加鲁棒性(Heikki Linnakangas)

  修复竞争条件和可能文件描述符泄露。

- 如果给定错误编码数据，并且 `client_encoding` 设置是客户端编码，比如SJIS， 修复psql中可能崩溃(Jiang Guiqing)

- 修复在 `tar` 输出格式中通过pg_dump发出的 `restore.sql` 脚本中错误(Tom Lane)

  该脚本可能在名字包含大小写字母的表中完全失败。 同时，使得该脚本可以恢复 `--inserts` 模式以及规则COPY模式中的数据。

- 修复pg_restore以接受符合POSIX标准的 `tar` 文件(Brian Weaver, Tom Lane)

  pg_dump的 `tar` 输出模式的原编码产生不完全符合POSIX标准的文件。 这在9.3版本已得到纠正。该补丁更新之前的分支，以致于他们同时接受不正确和校正后的格式， 当9.3出版的时候，希望避免兼容性问题。

- 当给定一个相对路径到数据目录的时候， 修复pg_resetxlog以正确定位 `postmaster.pid` (Tom Lane)

  该错误可能导致pg_resetxlog没有注意到有一个使用数据目录的活跃postmaster。

- 修复libpq的 `lo_import()` 和 `lo_export()` 函数以正确报告文件I/O错误(Tom Lane)
- 修复嵌套结构指针变量的ecpg的处理(Muhammad Usama)
- 当检查页的时候， 使得 `contrib/pageinspect` 的btree页检查函数采取缓冲锁(Tom Lane)
- 修复pgxs支持在AIX上建立可加载模块(Tom Lane)

  在原始树外部建构模块无法在AIX上运行。

- 为了Cuba, Israel, Jordan, Libya, Palestine, Western Samoa以及Brazil部分 中DST变化更新时区数据文件到tzdata发布2012j。

E.56. 发布8.3.21

> 发布日期: 2012-09-24

该发布包含来自8.3.20中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

PostgreSQL社区在2013年2月为8.3.X发布系列停止发布更新。 鼓励用户尽快更新到新版本分支。

E.56.1. 迁移版本8.3.21

运行8.3.X不需要备份/恢复。

然而，如果从8.3.17更早版本更新，参阅8.3.17发布说明。

E.56.2. 变化

- 提高GiST索引中页分离决定(Alexander Korotkov,Robert Haas, Tom Lane)

  多列GiST索引可能遭受到意想不到的膨胀，由于这个错误。

- 如果仍然持有权限，那么修复级联权限撤销中断(Tom Lane)

  如果我们从某个角色 `_X_` 中撤销授权选项， 但 `_X_` 仍然持有通过从其它人授权的选项， 否则，我们不应该递归撤销从 `_X_` 授予它的角色(s) `_Y_` 的相应权限。

- 当PL/Perl在使用中的时候，修复 `SIGFPE` 的处理(Andres Freund)

  Perl重置进程的 `SIGFPE` 处理器到 `SIG_IGN`， 这可能会随后导致崩溃。 在初始化PL/Perl之后恢复正常的Postgres信号处理器。

- 如果重新定义递归PL/Perl函数而被执行，避免PL/Perl崩溃(Tom Lane)

- 解决PL/Perl中可能的失败最佳化(Tom Lane)

  一些Linux发行版包含导致PL/Perl中不正确编译代码的 `pthread.h` 不正确的版本， 如果PL/Perl函数调用抛出错误的另外一个，那么导致崩溃。

- 为Fiji中DST变化更新时区数据文件到tzdata发布2012f。

E.57. 发布8.3.20

> 发布日期: 2012-08-17

该发布包含来自8.3.19各种修复，关于8.3主要发布中新特性信息， 参阅Section E.77。

PostgreSQL社区在2013年2月为8.3.X发布系列停止发布更新。 鼓励用户尽快更新到新版本分支。

E.57.1. 迁移到版本8.3.20

运行8.3.X不需要备份/恢复。

然而，如果从8.3.17更早版本更新，参阅8.3.17发布说明。

E.57.2. 变化

- 防止通过XML实体引用访问外部文件/URL(Noah Misch, Tom Lane)

  `xml_parse()`会试图获取外部文件或作为需要解决DTD和XML值中实体引用的URL， 从而使未经授权的数据库用户试图获取带有数据库服务器权限的数据。 当外部数据不会直接返回给用户的时候，它的部分可以暴露在错误信息中， 如果数据没有解析为有效的XML;并且在任何情况下检查文件存在的单纯能力可能对攻击者有用。

- 避免通过`contrib/xml2`的`xslt_process()`访问外部文件/URL(Peter Eisentraut)

  libxslt提供了通过样式表命令文件和URL读取和写入能力， 从而使未经授权的数据库用户读取和写入具有数据库服务器的权限数据。 禁用正确使用libxslt的安全选项。(CVE-2012-3488)

  此外，删除`xslt_process()`从外部文件/URL中获取文件和样式表。 当这是一个记录"特性"的时候， 长期被视为一个坏主意。 为CVE-2012-3489的修复中断这种能力，而非耗费精力试图修复它，我们只是简单将其删除。

- 防止过早回收利用btree索引页(Noah Misch)

  当我们允许只读事务跳过指定XID的时候，我们 引入一种可能性即一个已删除btree页可能被循环利用，而只读事务还在运行中。 这会导致不正确的索引搜索结果。发生在该字段中这种错误的可能性似乎非常低， 因为定时要求，但尽管如此它应该被修复。

- 修复新创建或者重置序列的崩溃安全错误(Tom Lane)

如果 `ALTER SEQUENCE` 在新创建或重置序列中被执行， 然后精确执行 `nextval()` 调用， 之后服务器崩溃，WAL重播将恢复序列到一个状态，它似乎没有 `nextval()` 被完成， 因此允许第一个序列值通过下一个 `nextval()` 调用再次被返回。 特别是，这可能表现为 `serial` 列， 因为串行列的序列的创建包括 `ALTER SEQUENCE OWNED BY` 步骤。

- 确保 `backup_label` 文件在 `pg_start_backup()` 之后 是fsync的(Dave Kerr)

- 回到补丁9.1改进以压缩fsync请求队列(Robert Haas)

  这提高了检查点期间性能。9.1变化已经有足够字段测试对于备份补丁是安全的。

- 仅仅允许autovacuum通过直接阻断进程被自动取消(Tom Lane)

  原编码可能允许某些情况下不一致操作；尤其是，autovacuum在小于 `deadlock_timeout` 宽限期后可能被取消。

- 提高autovacuum取消的记录(Robert Haas)

- 修复日志收集器以致于服务器启动后第一个日志循环期间 `log_truncate_on_rotation` 可以运行(Tom Lane)

- 确保子查询的整行引用不包含任何额外的 `GROUP BY` 或者 `ORDER BY` 列(Tom Lane)

- 不允许 `CHECK` 限制中拷贝整行引用以及 `CREATE TABLE` 中索引定义(Tom Lane)

  这种情况可以产生于具有 `LIKE` 或者 `INHERITS` 的 `CREATE TABLE` 中。 拷贝的整个行变量被错误地标记为原表的行类型不是新的。 拒绝该种情况对 `LIKE` 似乎是合理的， 因为该行类型很可能之后存在分歧， 为 `INHERITS` 我们理论上允许它，带有隐含强制父表行类型；但是这比似乎安全的备份补丁需要更多的工作。

- 修复 `ARRAY(SELECT ...)` 子查询中内存泄露(Heikki Linnakangas, Tom Lane)

- 修复从正则表达式中常见前缀的提取(Tom Lane)

  该代码可以通过量化括号子表达式被混淆，比如 `^(foo)?bar` 。 这可能导致搜索这种模式的不正确索引优化。

- 正确报告 `contrib/xml2` 的 `xslt_process()` 中的错误(Tom Lane)

- 为Morocco和Tokelau中DST变化更新时区数据文件到tzdata发布2012e。

E.58. 发布8.3.19

> 发布日期: 2012-06-04

该发布包含来自8.3.18各种修复，关于8.3主要发布中新特性信息， 参阅Section E.77。

E.58.1. 迁移到版本8.3.19

运行8.3.X不需要备份/恢复。

然而，如果从8.3.17更早版本更新，参阅8.3.17发布说明。

E.58.2. 变化

- 修复 `contrib/pgcrypto` 的DES `crypt()` 函数中 不正确密码转换(Solar Designer)

  如果密码字符串包含字节值 `0x80` ， 密码的剩余部分被忽略，从而导致该密码比它出现的更弱。 有了该修复，字符串的其余部分被正确包含在DES哈希中。 任何受此漏洞影响的存储密码值将因此不再匹配， 所以存储的值可能需要被更新。

- 为程序语言的调用处理器忽略 `SECURITY DEFINER` 和 `SET` 属性(Tom Lane)

  将这些属性应用到调用处理器可能使服务器崩溃(CVE-2012-2655)

- 允许 `timestamp` 输入中数字时区偏移量达到距离UTC 16小时(Tom Lane)

  一些历史时区偏移量超出之前限制15小时。这可能导致重载期间备份数据值被拒绝。

- 当给定时间恰好是当前时区最后DST过渡时间时，修复时间戳转换处理(Tom Lane)

  这种监督已经很长时间了，但是之前没有注意到，因为大部分DST使用区域被假定具有未来DST转换的无限序列。

- 修复 `text` 到 `name` 以及 `char` 到 `name` 映射 用来在多字节编码中正确执行字符串截断(Karl Schnaitter)

- 修复 `to_tsquery()` 中内存拷贝错误(Heikki Linnakangas)

- 当 `pg_attribute` 比较大的时候，修复缓慢会话启动(Tom Lane)

  如果 `pg_attribute` 超出 `shared_buffers` 的四分之一， 在会话启动期间需要缓存重建代码可能触发同步扫描逻辑。 导致它需要比正常长好几倍的时间。如果立刻启动许多新会话，问题尤其严重。

- 确保顺序扫描合理检查查询取消(Merlin Moncure)

  包含非活跃元组的许多连续页扫描不会相应中断。

- 确保 `PGSemaphoreLock()` 的Windows实现在返回前清理 `ImmediateInterruptOK` (Tom Lane)

  该疏忽意味着在相同的查询之后收到的查询取消中断 可以在一个不安全的时刻被接受，具有不可预测性，但没有好的结果。

- 当输出视图或者规则的时候，安全显示整行变量(Abbas Butt, Tom Lane)

  涉及模糊名字（也就是说，这个名字可以是表或者查询的列名字）的情况以模糊方式被输出， 存在风险是视图或者规则在备份或者重载之后可能被不同地解释。 通过附加非空操作转换避免模糊情况。

- 确保autovacuum工作进程正确执行堆栈深度检查(Heikki Linnakangas)

  之前，通过自动 `ANALYZE` 激发的函数中无限递归可能使得 工作进程崩溃。

- 修复日志收集器使得在高负载下不会丢失日志一致性(Andrew Dunstan)

  如果它比较繁忙，该收集器之前可能无法重新分配大信息。

- 修复日志收集器以确保它在收到SIGHUP之后重启文件循环(Tom Lane)

- 当该目标是函数的第一个变量的时候， 修复PL/pgSQL的 `GET DIAGNOSTICS` 命令(Tom Lane)

- 当数据库包含多个对象的时候，修复pg_dump中若干个性能问题(Jeff Janes, Tom Lane)

  如果数据库包含许多模式，或者如果许多对象在依赖循环中，或者如果有许多拥有的序列， 那么pg_dump可能会很慢。

- 修复 `contrib/dblink` 的 `dblink_exec()` 而不泄露 临时数据库连接错误(Tom Lane)

- 为Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria和Tokelau Islands中DST变化更新 时区数据文件到tzdata发布2012c； 同时为Canada历史修正。

E.59. 发布8.3.18

> 发布日期: 2012-02-27

该发布包含来自8.3.17各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.59.1. 迁移到版本8.3.18

运行8.3.X不需要备份/恢复。

然而，如果你从8.3.17更早版本更新，参阅8.3.17发布说明。

E.59.2. 变化

- 为 `CREATE TRIGGER` 需要触发器函数执行权限(Robert Haas)

  这种缺失检查可能允许其他用户执行具有虚假输入数据的触发器函数， 通过所拥有的表上进行安装。 对于触发器函数标记 `SECURITY DEFINER` 是有意义的， 因为否则作为表的所有者运行触发器函数。

- 转换换行到写入pg_dump中名字中的空格(Robert Haas)

  pg_dump有关审查其输出脚本中SQL注释中发出的对象名称是不谨慎的。 包含新行的名字至少会呈现脚本语法不正确。 当脚本被重新加载的时候， 恶意制作的对象名称可以提出一个SQL注入风险。

- 修复来自清理插入中的btree索引损坏(Tom Lane)

  插入造成的索引页拆分有时可能会导致 同时运行 `VACUUM` 错过删除它应该删除的索引项。 对应表中的行被删除后， 悬空索引项会导致错误（如"不能读取文件块N..."）， 或者更糟的是，错误查询结果 在不相关行之后重新被插入到已释放表位置。 此bug已自8.2版本中存在， 但发生如此频繁直到现在并没有被诊断。 如果您有理由怀疑它出现在你的数据库中，重新索引受影响的索引会修复该问题。

- 允许 `ALTER USER/DATABASE SET` 中一些设置的非存在值(Heikki Linnakangas)

  允许 `default_text_search_config` ， `default_tablespace` 和 `temp_tablespaces` 设置为那些不知道的名字。 这是因为它们可能在另一个打算使用该设置的数据库中是已知的，或者表空间中， 因为该表空间可能尚未被创建。同样的问题之前公认为 `search_path` ， 并且这些设置现在像那一个。

- 正确跟踪WAL回放中OID计数器，即使它绕回(Tom Lane)

之前的OID计数器将继续保持高值，直到系统退出重放模式。 实际结果通常是零，但是存在这种情况，其中备用服务器被提升为主服务器可能需要较长的时间 推进OID计数器到一个合理的值，一旦需要该值的时候。

- 修复附加 `*` 的正则表达式反向引用(Tom Lane)

  不是强制执行精确字符串匹配， 代码将有效地接受任何满足引用反向引用符号的模式子表达式的字符串。

  类似的问题仍然困扰嵌入到较大量化表达式中的反向引用， 而不是量词的直接主题。 这将在未来的PostgreSQL版本中得到解决。

- 修复在处理 `inet` / `cidr` 值中最近引入的内存泄漏(Heikki Linnakangas)

  在2011年12月发布PostgreSQL补丁导致操作中内存泄漏， 这可能在下面情况中非常重要，比如在这样的列中构建btree索引。

- 避免Windows上syslogger中文件句柄的双关闭(MauMau)

  通常这种错误是不可见的，但是当运行Windows的调试版本时，它可能会导致异常。

- 修复plpgsql中I/O转换相关内存泄露(Andres Freund, Jan Urbanski, Tom Lane)

  某些操作可能会造成内存泄漏，直到当前函数结束。

- 提高继承表列的pg_dump处理(Tom Lane)

  pg_dump处理不当的情况下，子列比其父列有不同的缺省表达式。 如果缺省等同于父缺省，但实际上并不是相同的（比如，因为模式搜索路径差异），它可能 不会被识别为不同的，以便转储之后恢复子列可能被允许继承父缺省。 子列是 `NOT NULL` ，其中父列不是的也可以错误地被恢复。

- 为INSERT形式表数据修复pg_restore指向数据库模式(Tom Lane)

  当使用发布日期为9月或2011年12月份的pg_restore的时候， 直接到数据库从采用 `--inserts` 或者 `--column-inserts` 选项的归档文件恢复失败。 作为另一个问题修复的监督结果。 归档文件本身没有过错，而且文本模式输出是好的。

- 修复 `contrib/intarray` 's `int[] & int[]` 操作符中的错误(Guillaume Lelarge)

  如果最小整数的两个输入数组的共同点是1，并且在任何数组中有较小的值， 那么1将被错误地从结果中忽略。

- 修复 `contrib/pgcrypto` 的 `encrypt_iv()` 和 `decrypt_iv()` 中的错误检测(Marko Kreen)

  这些函数没有报告无效输入错误的一些类型， 而是返回不正确输入的随机的垃圾值。

- 修复 `contrib/test_parser` 中一个字节缓冲区溢出(Paul Guyot)

该代码试图读取一个比它应该更多的字节，在极端情况中可能崩溃。 由于`contrib/test_parser`只是示例代码， 这本身并不是一个安全问题，但糟糕的示例代码仍然很糟糕。

- 如果可用，在ARM上为自旋锁使用`__sync_lock_test_and_set()`(Martin Pitt)

  这个函数替代了我们先前使用的`SWPB`指令，它被废弃了， 并且在ARMv6及更高版本中不可用。报告表明旧的代码在最近的ARM上以明显的方式执行， 但根本没有互锁并发访问，导致多进程操作离奇故障。

- 当使用接受它的gcc版本构建的时候，使用`-fexcess-precision=standard`选项(Andrew Dunstan)

  这可以防止各类情况，其中最近的gcc版本将产生创造性结果。

- 允许FreeBSD上线程Python的使用(Chris Rees)

  我们的配置脚本之前认为这个组合不会正常工作； 但FreeBSD解决了这一问题，所以删除错误检查。

E.60. 发布8.3.17

> 发布日期: 2011-12-05

该发布包含来自8.3.16中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.60.1. 迁移到版本8.3.17

运行8.3.X不需要备份/恢复。

然而，在 `information_schema.referential_constraints` 视图中发现长期的错误。 如果你依赖于该视图中正确的结果，那么你应该像下面第一个更新日志项解释的来代替它的定义。

另外，如果你从8.3.8更早版本更新，参阅8.3.8发布说明。

E.60.2. 变化

- 修复 `information_schema.referential_constraints` 视图中错误(Tom Lane)

  该视图关于外键约束匹配依赖主或者唯一键约束是不够谨慎的， 这可能导致无法显示外键约束，或显示它多次，或声称它 取决于比起真正的一个不同的约束。

  由于视图定义是由initdb安装， 只是升级不会解决问题。如果您需要解决现有的安装中这个问题， 您可以（作为超级用户）删除 `information_schema` 模式，然后通过 `_SHAREDIR_` /information*schema.sql*重新创建它。 （如果你不确定 `_SHAREDIR* `在哪里， 运行` pg_config --sharedir`）。在每个数据库中肯定重复被修复。

- 修复 `CREATE TABLE dest AS SELECT * FROM src` 或者 `INSERT INTO dest SELECT * FROM src` 中 TOAST相关数据损坏(Tom Lane)

  如果一个表已由 `ALTER TABLE ADD COLUMN` 修改， 尝试逐字拷贝其数据到另一个表可能在某些极端情况中产生损坏结果。 这个问题只能体现在8.4及更高版本中的精确形式中， 但是我们修补早期版本以及其他的代码路径中可能会引发同样的错误。

- 修复toast表访问旧的syscache项中的竞争条件(Tom Lane)

  典型的症状是类似"在pg_toast_2619中toast值NNNNN缺少块号0"的短暂错误， 其中所引用的toast表将永远属于一个系统目录。

- 让 `DatumGetInetP()` 解压具有1字节头的inet数据， 并添加一个新的宏， `DatumGetInetPP()` 不这样做(Heikki Linnakangas)

这种变化不会影响到核心代码， 但可能会阻止附加代码预期 `DatumGetInetP()` 按通常惯例产生解压缩数据而崩溃。

- 提高 `money` 类型输入和输出中的区域支持(Tom Lane)

  除了不支持所有标准 `lc_monetary` 格式选项外， 输入和输出函数是不一致的， 意味着有个区域中备份的 `money` 值不能被重新读取。

- 不要让 `transform_null_equals` 影响 `CASE foo WHEN NULL ...` 结构(Heikki Linnakangas)

  `transform_null_equals` 应该影响由用户直接写入的 `foo = NULL` 表达式， 而不是通过 `CASE` 形式内部产生的平等检查。

- 改变外键触发器创建顺序更好地支持自参考外键(Tom Lane)

  对于级联外键引用自己的表， 行更新将触发 `ON UPDATE` 触发器和 `CHECK` 触发器作为一个事件。 `ON UPDATE` 触发器必须先执行， 否则 `CHECK` 将检查该行的非最终状态而且可能引发不当的错误。 然而，这些触发器的触发顺序由其名字确定， 通常按照创建顺序进行排序， 因为触发器有按照约定"RI_ConstraintTrigger_NNNN"自动生成的名字。 一个适当的修复要求修改该约定， 我们将在9.2中执行，但在现有版本中改变它似乎有风险。 所以这个补丁只是改变了触发器的创建顺序。用户遇到此类型的错误 要删除并重新创建外键约束使得触发器到正确的顺序。

- 当跟踪缓冲区分配率时，避免浮点下溢(Greg Matthews)

  虽然本身无害，在某些平台上这可能导致恼人的内核日志信息。

- 保留psql命令历史中空白行(Robert Haas)

  比如，如果空行从一个字符串中被删除，前者操作可能会导致问题。

- 修复pg_dump以备份自动生成类型之间用户定义的映射， 比如表行类型(Tom Lane)

- 使用xsubpp的首选版本构建PL/Perl， 不一定是操作系统主拷贝(David Wheeler和Alex Hunsaker)

- 修复 `contrib/dict_int` 和 `contrib/dict_xsyn` 中不正确编码(Tom Lane)

  一些函数错误假定通过 `palloc()` 返回的内存是零。

- 实现 `pgstatindex()` 中查询取消及时中断(Robert Haas)

- 确保VPATH构造正确安装所有服务器头文件(Peter Eisentraut)

- 缩短详细错误信息中报告的文件名(Peter Eisentraut)

  常规构造一直报道包含错误信息调用的C文件名字，但VPATH构造之前报道绝对路径名。

- 修复中美洲地区Windows时区名称解释(Tom Lane)

映射"Central America Standard Time"到`CST6`，而不是`CST6CDT`，因为在中美洲任何地方通常不会观察到DST。

- 为Brazil, Cuba, Fiji, Palestine, Russia和Samoa更新时区数据文件到tzdata发布2011n；同时为Alaska和British East Africa历史修正。

E.61. 发布8.3.16

> 发布日期: 2011-09-26

该发布包含来自8.3.15中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.61.1. 迁移到版本8.3.16

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.61.2. 变化

- 修复存在疑问HOT更新元组的索引中错误(Tom Lane)

  该错误导致在重新索引系统目录之后索引损坏。 他们不认为会影响用户索引。

- 修复GiST索引页划分处理中多个错误(Heikki Linnakangas)

  发生的概率很低，但是这些可能导致索引损坏。

- 修复 `tsvector_concat()` 中可能的缓冲区溢出(Tom Lane)

  该函数可能低估了该结果需要内存量，导致服务器崩溃。

- 当处理"standalone"参数的时候，修复 `xml_recv` 中崩溃(Tom Lane)

- 可能避免 `ANALYZE` 中以及SJIS-2004编码转换中访问内存结尾(Noah Misch)

  这修复了一些概率较低的服务器崩溃情况。

- 修复relcache初始文件失效中竞态条件(Tom Lane)

  有一个窗口，其中一个新的后端进程可以读取一个陈旧的启动文件， 但错过了可能告诉它数据是陈旧的无效消息。 其结果在目录访问中可能是古怪错误，通常在启动过程之后"无法读取文件中块0..."。

- 修复GiST索引扫描结尾内存泄露(Tom Lane)

  执行许多单独的GiST索引扫描命令，比如 已经含有许多行的表中基于排斥约束新的GiST验证， 由于该泄露，可能瞬时需要大量内存。

- 当构建大的，有损耗位图的时候，修复性能问题(Tom Lane)

- 修复数组和路径创建函数以确保填充字节为零(Tom Lane)

这避免了规划器将认为语义相同常数是不相等的，导致劣质优化。

- 解决打断WAL回放gcc 4.6.0错误(Tom Lane)

  在服务器崩溃后这可能导致已提交事务丢失。

- 为视图中 `VALUES` 修复备份错误(Tom Lane)
- 不允许序列上 `SELECT FOR UPDATE/SHARE` (Tom Lane)

  该操作不能像预期那样执行，并且可能导致错误。

- 当计算哈希表大小的时候，防御整数溢出(Tom Lane)
- 修复 `CLUSTER` 可能尝试访问已删除TOAST数据的情况(Tom Lane)
- 修复为"peer"认证使用凭证控制信息中的可移植错误(Tom Lane)
- 当需要多个往返时，修复SSPI登录(Ahmed Shinwari,Magnus Hagander)

  这个错误典型现象是在SSPI登录期间"不支持函数需求"错误。

- 修复 `pg_srand48` 开始初始化中的错误(Andres Freund)

  这导致未能使用提供了种子的所有位。 这个函数在大多数平台上（只有那些没有 `srandom` ）没有被使用， 并且暴露于比预期种子不太随机的潜在安全隐患在任何情况下似乎很小。

- 当 `LIMIT` 和 `OFFSET` 值总和超过2^63的时候，避免整数溢出(Heikki Linnakangas)
- 添加溢出检查到 `generate_series()` 的 `int4` 和 `int8` 版本(Robert Haas)
- 修复 `to_char()` 中尾随零删除(Marti Raudsepp)

  在 `FM` 格式中，小数点后没有数字位置，零到小数点左边可能错误地被删除。

- 修复 `pg_size_pretty()` 避免输入接近2^63溢出(Tom Lane)
- 在pg_ctl中，为Windows上服务注册支持静止模式(MauMau)
- 在从一个不同文件进行 `COPY` 期间修复脚本文件行的psql的计算(Tom Lane)
- 为 `standard_conforming_strings` 修复pg_restore直接到数据库模式(Tom Lane)

  当从 `standard_conforming_strings` 设置为 `on` 的归档文件直接恢复到数据库服务器时，pg_restore可能会发出不正确的命令。

- 修复写缓冲结束以及libpq的LDAP服务查找代码中的内存泄露(Albe Laurenz)
- 在libpq中，当使用非阻塞I/O以及SSL连接时，避免错误(Martin Pihlak, Tom Lane)
- 在连接启动过程中优化错误的libpq处理(Tom Lane)

尤其是，在SSL连接启动过程中 `fork()` 错误的服务器报告回应是明智的。

- 为SSL错误优化libpq的错误报告(Tom Lane)
- 采用ecpglib书写带有15位精度的 `double` 值(Akira Kurosawa)
- 在ecpglib中，确保 `LC_NUMERIC` 设置在错误之后被恢复(Michael Meskes)
- 为加密符号字符错误(CVE-2011-2483)请求上游修复(Tom Lane)

  `contrib/pg_crypto` 的加密代码可能给定错误结果，在char是有符号平台上（这是大多数情况），导致加密密码比他们应该的更弱。

- 修复 `contrib/seg` 中内存泄露 (Heikki Linnakangas)
- 修复 `pgstatindex()` 为空索引给定一致结果(Tom Lane)
- 允许使用perl 5.14编译(Alex Hunsaker)
- 为探测系统函数的存在更新配置脚本方法(Tom Lane)

  在8.3和8.2中使用的autoconf的版本可能通过执行链接时优化编译器被愚弄。

- 修复编译和安装包含空格的文件路径的各种错误(Tom Lane)
- 为Canada, Egypt, Russia, Samoa和South Sudan中DST变化更新时区数据文件到 tzdata 发布2011i。

E.62. 发布8.3.15

> 发布日期: 2011-04-18

该发布包含来自8.3.14的各种修复。关于8.3主要发布新特性的信息， 参阅Section E.77。

E.62.1. 迁移到版本8.3.15

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.62.2. 变化

- 不允许完全包含复合类型(Tom Lane)

  当处理复合类型的时候，这可以防止服务器可能无限递归的情况。 虽然有可能用于这样的结构，他们似乎并没有足够的说服力来证明需要努力确保它始终安全工作。

- 在目录缓存初始化期间避免潜在死锁(Nikhil Sontakke)

  在某些情况下，缓存加载代码将在锁定索引目录之前系统索引上获取共享锁。 这可能会死锁进程试图获取其他排它锁，以更标准顺序。

- 当有一个并发更新到目标元组的时候，修复`BEFORE ROW UPDATE`触发器处理中 悬垂指针问题(Tom Lane)

  当尝试执行`UPDATE RETURNING ctid`的时候， 该错误已被观察导致间歇性"不能提取虚拟元组系统属性"故障。 对于更严重错误有一个非常小的概率，比如为更新元组产生不正确索引项。

- 当该表有等待延迟触发器事件的时候，不允许`DROP TABLE` (Tom Lane)

  之前`DROP`可能完成，当触发器被最终触发的时候， 导致"不能打开带有OID nnn的关系"错误。

- 修复涉及到数组片段的PL/Python内存泄露(Daniel Popowich)

- 修复pg_restore以处理TOC文件中长行（超过1KB）(Tom Lane)

- 采取更多保障措施防止崩溃，由于使用编译器优化除以零(Aurelien Jarno)

- 支持在MIPS上FreeBSD和OpenBSD中使用dlopen()(Tom Lane)

  有一个硬连线假设该系统函数不可用在这些系统上的MIPS硬件上。 相反使用编译时间测试，因为更多新版本拥有它。

- 修复HP-UX上编译错误(Heikki Linnakangas)
- 修复Windows上与libintl版本兼容问题(Hiroshi Inoue)
- 修复Windows编译脚本中xcopy的用法以便在Windows 7中正确执行(Andrew Dunstan)

  这只影响编译脚本，而不是安装或者用法。
- 在Cygwin上通过pg_regress来修复路径分隔符(Andrew Dunstan)
- 为了Chile, Cuba, Falkland Islands, Morocco, Samoa和Turkey中DST变化更新时区数据文件到tzdata发布2011f；同时为了South Australia, Alaska和Hawaii历史修正。

E.63. 发布8.3.14

> 发布日期: 2011-01-31

该发布包含来自8.3.13中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.63.1. 迁移到版本8.3.14

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.63.2. 变化

- 当 `EXPLAIN` 尝试显示简单形式 `CASE` 表达式的时候， 避免错误(Tom Lane)

  如果 `CASE` 的测试表达式是一个常量，那么规划器可以简化 `CASE` 到迷惑表达式显示代码的一种形式中， 导致"意外CASE WHEN子句"错误。

- 修复下标现有范围之前数组片段的分配(Tom Lane)

  如果在新添加的下标和最初预先存在的下标之间有间隙， 那么代码误算需要从旧数组空位图中拷贝多少项， 有可能导致数据损坏或崩溃。

- 避免规划遥远日期值中意外转换溢出(Tom Lane)

  `date` 类型支持比 `timestamp` 类型表示的更广泛的日期， 但规划器假定它可能总是转换日期到timestamp。

- 当 `standard_conforming_strings` 是on的时候， 修复大对象(BLOBs)pg_restore的文本输出(Tom Lane)

  虽然直接恢复到数据库正常执行，如果pg_restore被要求 SQL文本输出并且源数据库中已经启用 `standard_conforming_strings` ， 那么字符串逃避是不正确的。

- 修复包含 `... & !(subexpression) | ...` 的 `tsquery` 值 的错误解析(Tom Lane)

  不能正确执行包含操作符合并的查询。 `contrib/intarray` 的 `query_int` 类型和 `contrib/ltree` 的 `ltxtquery` 类型中存在同样的错误。

- 为 `query_int` 类型修复 `contrib/intarray` 的输入函数中的缓冲区溢出(Apple)

  该错误是一个安全风险，因为函数的返回地址可能被重写。感谢Apple Inc的安全团队报告这个问题 并且提供修复(CVE-2010-4015)

- 修复 `contrib/seg` 的GiST picksplit算法错误(Alexander Korotkov)

这可能导致效率低下，尽管不是真的错误答案，在`seg`列的GiST索引中。 如果你有一个索引，在安装这个更新之后考虑`REINDEX`它。 （这与之前更新中`contrib/cube`中修复的错误是一样的）

E.64. 发布8.3.13

> 发布日期: 2010-12-16

该发布包含来自8.3.12的各种修复。关于8.3主要发布的新特性信息，参阅 Section E.77。

E.64.1. 迁移到版本8.3.13

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.64.2. 变化

- 强迫缺省 `wal_sync_method` 为Linux上 `fdatasync` (Tom Lane, Marti Raudsepp)

  Linux上的缺省很多年一直 `fdatasync` ， 但最近的内核变化反而引起PostgreSQL到选择 `open_datasync` 。 这一选择不会引起任何性能改进，并且在某些文件系统造成彻底的失败， 尤其是带有 `data=journal` 安装选项的 `ext4` 。

- 修复GIN索引WAL回放逻辑中各种错误(Tom Lane)

  这可能导致"bad buffer id: 0"错误或者复制过程中索引内容的损坏。

- 当开始检查点WAL记录不在同一WAL段作为重做点，修复基础备份恢复(Jeff Davis)

- 当多个执行者长时间保持活跃，修复autovacuum持久延迟(Tom Lane)

  为自动清理者的有效 `vacuum_cost_limit` 可能会删除到几乎为零， 如果它处理了足够的表，导致它运行得非常慢。

- 添加支持检测 `IA64` 上寄存器堆栈溢出(Tom Lane)

  `IA64` 结构有两个硬件堆栈。 全面预防堆栈溢出错误需要检查两者。

- 添加 `copyObject()` 中的堆栈溢出检查(Tom Lane)

  某些代码路径可能崩溃，因堆栈溢出产生一个足够复杂的查询。

- 修复临时GiST索引中页面拆分检查(Heikki Linnakangas)

  它可能在临时索引中有一个"并发"页拆分， 例如，当插入完成的时候，如果有一个打开的游标扫描索引， 当游标继续执行的时候，GiST未能检测这种情况，并因此可能提供错误的结果。

- 当 `ANALYZE` 复杂索引表达式的时候，避免内存泄露(Tom Lane)

- 确保使用整行Var的索引仍然依赖于它的表(Tom Lane)

  当删除表的时候，声明像 `create index i on t (foo(t.*))` 的索引可能不会自动被删除。

- 不能"内联"SQL函数与多个 `OUT` 参数(Tom Lane)

  这避免可能崩溃，由于关于预期结果行类型信息缺失。

- 如果 `ORDER BY` ， `LIMIT` ， `FOR UPDATE` 或者 `WITH` 附属于 `INSERT ... VALUES` 的 `VALUES` 部分，则操作正确(Tom Lane)

- 修复 `COALESCE()` 表达式的常数合并(Tom Lane)

  规划器可能有时尝试评估实际上可能找不到的子表达式，可能导致意外错误。

- 当连接尝试（ `accept()` 或者它之后立即调用）失败时， postmaster与GSSAPI支持一起编译(Alexander Chernikov)

- 当 `log_temp_files` 是活跃的，修复临时文件的忽略断开(Tom Lane)

  当尝试发出日志信息的时候，如果发生错误，那么删除链接不执行，导致临时文件积累。

- 添加 `InhRelation` 节点输出功能(Tom Lane)

  当激活 `debug_print_parse` ，并且执行某种类型查询的时候，避免错误。

- 修复从一个点到一个水平线段不正确计算(Tom Lane)

  该错误影响了若干个不同几何距离测量操作符。

- 修复"simple"表达式的PL/pgSQL处理可以在 递归或者错误恢复情况中执行(Tom Lane)

- 修复集合返回函数的PL/Python的处理(Jan Urbanski)

  在迭代器产生一组结果中尝试调用SPI函数可能失败。

- 修复 `contrib/cube` 的GiST picksplit算法中的错误(Alexander Korotkov)

  这可能会导致相当大的效率低下，但实际上并不是错误结论，在GiST索引 `cube` 列中。如果你有这样一个索引，考虑在安装此更新后 `REINDEX` 它。

- 不能发出 `contrib/dblink` 中"标识符被截断"注意事项，除非 当创建新链接的时候(Itagaki Takahiro)

- 修复 `contrib/pgcrypto` 中关于丢失公钥上潜在核心转储(Marti Raudsepp)

- 修复 `contrib/xml2` 的XPath查询函数中的内存泄露(Tom Lane)

- 为了Fiji和Samoa中DST变化更新时区数据文件到tzdata发布2010o； 同时为了Hong Kong历史修正。

E.65. 发布8.3.12

> 发布日期: 2010-10-04

该发布包含来自8.3.11的各种修复。 关于8.3主要发布的新特性信息，参阅Section E.77。

E.65.1. 迁移到版本8.3.12

为运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.65.2. 变化

- 为PL/Perl和PL/Tcl中每个调用SQL userid使用单独解析器 (Tom Lane)

  这种变化可以防止通过颠覆在同一会话下另一个SQL用户身份执行的 （例如， `SECURITY DEFINER` 函数内）Perl或Tcl的代码引起的安全问题。 大多数的脚本语言提供可以执行的多种方式，比如重新定义通过目标函数调用的标准函数或操作符。 如果没有这些变化，使用Perl或Tcl语言使用权的任何SQL用户本质上可以执行 目标函数所有者的SQL权限。

  这种变化的成本是在Perl和Tcl函数之间有意通讯变得更加困难。 为了提供一个安全出口，PL/PerlU和PL/TclU函数继续使用每个会话中解析器。 没有考虑安全问题，因为所有这些函数在数据库超级用户的信任级别中执行。

  这很可能是第三方的程序语言，声称可以提供可信认执行程序也有类似的安全问题。 我们建议您联系依赖于安全关键目的的任何PL作者。

  我们非常感谢Tim Bunce指出这个问题(CVE-2010-3433)

- 避免在 `pg_get_expr()` 中通过禁用它 打算用于不是系统目录列中的参数的可能崩溃 (Heikki Linnakangas, Tom Lane)

- 把退出代码128(`ERROR_WAIT_NO_CHILDREN`)作为Windows上非致命的(Magnus Hagander)

  在高负载下，Windows进程有时会在错误代码启动时失败。 之前postmaster将这个看作为恐慌条件，并且重新启动整个数据库， 但是这似乎是一种过度反应。

- 修复附加索引扫描中非严格的OR 连接子句的不正确用法(Tom Lane)

  这是在8.3分支中丢失的8.4修复的一个补丁。 这纠正了8.3.8中引入的一个错误，它可能导致外连接不正确 结果，当内在关系是继承树或者 `UNION ALL` 子查询的时候。

- 修复 `UNION ALL` 成员关系的可能重复扫描(Tom Lane)

- 修复"不能处理非计划子查询"错误(Tom Lane)

  当子选择包含扩展到包含另一子选择的表达式中连接别名引用的时候发生。

- 修复该错误以标记缓存计划为瞬时的(Tom Lane)

  如果已准备好规划，当 `CREATE INDEX CONCURRENTLY` 是所引用表中之一， 它被认为是重新规划，一旦索引准备使用。这没有发生。

- 在一些偶然报告的btree错误情况中减少PANIC到ERROR，并且提供错误信息的更多细节(Tom Lane)

  这可能优化使用损坏索引系统鲁棒性。

- 防止show_session_authorization()在autovacuum进程中崩溃(Tom Lane)

- 防御函数返回setof记录，其中并非所有返回行都是同一行类型(Tom Lane)

- 当散列按引用传递函数结果的时候，修复可能错误(Tao Ma, Tom Lane)

- 优化join列中NULL的合并连接处理(Tom Lane)

  如果排序顺序是空排序高，那么合并连接完全达到第一个空时停止。

- 当写它们的时候，注意fsync锁文件的内容（包含 `postmaster.pid` 和套接锁文件）(Tom Lane)

  如果postmaster启动后不久主机崩溃，这种忽略可能会导致损坏的锁文件内容。 这可能反过来防止后续试图随后启动postmaster， 直到锁文件被手动删除。

- 当分配XID到大量嵌套子事务的时候，避免递归(Andres Freund, Robert Haas)

  如果有有限的堆栈空间，那么原代码可能导致崩溃。

- 避免在walwriter进程中保持打开旧的WAL段(Magnus Hagander, Heikki Linnakangas)

  之前代码可能阻止删除不需要段。

- 修复 `log_line_prefix` 的 `%i` 逃逸， 这可能在后端启动初期产生垃圾(Tom Lane)

- 当启动归档的时候，修复 `ALTER TABLE ... SET TABLESPACE` 中可能的数据损坏(Jeff Davis)

- 允许 `CREATE DATABASE` 和 `ALTER DATABASE ... SET TABLESPACE` 通过查询取消被中断(Guillaume Lelarge)

- 修复 `REASSIGN OWNED` 以处理操作符类和族(Asko Tiidumaa)

- 当比较两个空 `tsquery` 值的时候，修复可能的核心转储(Tom Lane)

- 修复包含 `%` 伴随 `_` 模式的 `LIKE` 的处理(Tom Lane)

我们之前已经修复这个问题，但是仍然有一些不正确处理情况。

- 在PL/Python中，防止来自 `PyCObject_AsVoidPtr` 和 `PyCObject_FromVoidPtr` 空指针结果 (Peter Eisentraut)

- 使得psql认为 `DISCARD ALL` 作为不应该包含在autocommit-off模式事务块中的命令(Itagaki Takahiro)

- 修复ecpg以处理正确来自 `RETURNING` 子句的数据(Michael Meskes)

- 完善包含已删除列的表的 `contrib/dblink` 的处理(Tom Lane)

- 修复 `contrib/dblink` 中"重复连接名"错误后的连接泄露(Itagaki Takahiro)

- 修复 `contrib/dblink` 以正确处理超过62字节的连接名(Itagaki Takahiro)

- 添加 `hstore(text, text)` 函数到 `contrib/hstore` (Robert Haas)

  该函数代替已过时的 `=>` 操作符。 它打补丁以便未来代码可用于旧的服务器版本。 请注意，该补丁在 `contrib/hstore` 被安装或重新安装在一个特定的数据库之后有效。 相反用户可能更愿意手动执行 `CREATE FUNCTION` <命令>。

- 更新编译基础设施和文档以反映从CVS到Git迁移的源代码库(Magnus Hagander and others)

- 为了Egypt和Palestine中的DST变化更新时区数据文件到tzdata发布2010l； 同时为了Finland历史修正。

  这种变化还增加了两个Micronesian时区新的名称： Pacific/Chuuk现在优于Pacific/Truk（并且首选缩写CHUT而非TRUT） Pacific/Pohnpei优于Pacific/Ponape。

- 采用Windows的"N. Central Asia标准时间"时区映射到Asia/Novosibirsk，而不是Asia/Almaty (Magnus Hagander)

  Microsoft改变了来自KB976098时区更新中该时区的DST操作。 Asia/Novosibirsk更好的匹配它的新操作。

E.66. 发布8.3.11

> 发布日期: 2010-05-17

该发布包含来自8.3.10的各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.66.1. 迁移到版本8.3.11

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新，参阅8.3.8发布说明。

E.66.2. 变化

- 在 `plperl` 中使用用于整个解释器的opmask强制限制， 而不是使用 `Safe.pm` (Tim Bunce, Andrew Dunstan)

  最近发展使我们确信 `Safe.pm` 太不安全而依靠 `plperl` 。 此变化删除了 `Safe.pm` 的使用，有利于使用应用于操作掩码的独立解析器。 该变化的副作用包括它在 `plperl` 中以一种自然的方式使用Perl的 `strict` 编译是可能的。 并且Perl的 `$a` 和 `$b` 变量在某种程序中按预期执行，而且函数编译显著更快。

- 防止PL/Tcl执行来自 `pltcl_modules` 的不可靠代码(Tom)

  从一个数据库表中自动加载的Tcl代码PL/Tcl功能可能会被特洛伊木马攻击利用， 因为关于谁可以创建或者插入该表没有限制。 这种变化禁用该功能，除非 `pltcl_modules` 是由超级用户拥有。 （但是，不检查表上的权限，真正需要一个低于安全模块表这样的安装仍然可以给予适当的权限到值得信赖的非超级用户）。 另外，防止代码加载到非限制"normal" Tcl解释器，除非我们真的要执行 `pltclu` 函数。(CVE-2010-1170)

- 如果在relcache项重建过程中收到缓存重置信息，那么修复可能崩溃(Heikki)

  当修复了相关错误时，该错误被引入到8.3.10中。

- 当为了该函数运行语言验证器的时候，应用每个函数GUC设置(Itagaki Takahiro)

  这就避免了失败，如果该函数的代码没有设置无效； 一个例子是SQL函数可能不解析，如果 `search_path` 是不正确的。

- 不允许无特权用户重置超级用户参数设置(Alvaro)

  以前，如果一个非特权用户为自身运行 `ALTER USER ... RESET ALL` ， 或者为拥有的数据库运行 `ALTER DATABASE ... RESET ALL` ， 这将为用户或数据库删除所有特殊参数设置， 甚至是那些只应该由超级用户改变的参数。 现在， `ALTER` 将只删除该用户有权限改变的

参数。

- 当 `CONTEXT` 附加物到日志项的时候， 如果发生关机，那么避免后台关机期间可能崩溃(Tom)

  在某些情况中上下文输出函数可能失败，因为当输出日志信息的时候，当前事务已经被回滚。

- 确保归档进程尽可能快地响应 `archive_command` 中变化(Tom)

- 为现代Perl版本更新pl/perl的 `ppport.h` (Andrew)

- 修复pl/python中各种内存泄露(Andreas Freund, Tom)

- 当扩展指向自身的一个变量的时候，避免psql中无限递归(Tom)

- 修复psql的 `\copy` 在 `\copy (select ...)` 中点周围 不添加空格(Tom)

  在数字文本小数点周围添加空格可能导致语法错误。

- 为使用 `contrib/intarray` 操作符的未能满足的查询修复 不必要"GIN索引不支持整个索引扫描"错误(Tom)

- 确保 `contrib/pgstattuple` 函数做出反应及时取消中断(Tatsuhito Kasahara)

- 使得服务器启动正确处理 为已存在共享内存段使得 `shmget()` 返回 `EINVAL` 的情况(Tom)

  这种操作已在BSD派生内核包括OS X中被观察。 这导致了一个完全误导启动失败，抱怨共享内存请求大小太大。

- 避免Windows上syslogger过程中可能崩溃(Heikki)

- 更加鲁棒性处理Windows注册表中不完整时区信息(Magnus)

- 更新已知的Windows时区名字设置(Magnus)

- 为Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia中DST变化更新 时区数据文件到tzdata发布2010j；同时为Taiwan历史修正。

  另外，添加 `PKST` (Pakistan Summer Time)到时区缩写的缺省设置中。

E.67. 发布8.3.10

> 发布日期: 2010-03-15

该发布包含来自8.3.9的各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.67.1. 迁移到版本8.3.10

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.67.2. 变化

- 添加新的配置参数 `ssl_renegotiation_limit` 以控制多久为SSL连接执行会话密钥协商(Magnus)

  这可以被设置为零，以完全禁用重新协商，如果使用损坏的SSL库，这可能是必需的。特别是，一些供应商为CVE-2009-3555导致谈判尝试失败传输临时补丁。

- 在后台启动期间修复可能死锁(Tom)
- 由于relcache重新载入期间不完全处理错误，那么修复可能崩溃(Tom)
- 由于悬空指针用于缓存计划中，修复可能崩溃(Tatsuo)
- 当试图从子事务启动中故障恢复的时候，修复可能崩溃(Tom)
- 修复与使用保存点和不同于服务器编码的客户端编码相关的服务器内存泄漏(Tom)
- 修复GIST索引页拆分的结束恢复清理过程中发出的不正确WAL数据(Yoichi Hirai)

  如果在已经完成一个不完整GIST插入后结束恢复清理期间不够幸运而发生崩溃， 这可能导致索引损坏，甚至更可能发生WAL回放期间的错误。

- 为 `bit` 类型采用 `substring()` 将任何负长度看作"所有其余字符串" (Tom)

  之前代码可以那样处理-1，并且为其它负数值产生无效结果值， 可能导致崩溃(CVE-2010-0442)。

- 当输出比特宽度大于不同于8位倍数的其它给定整数的时候， 修复整数到比特字符串转换以正确处理第一个分数字节(Tom)
- 修复异常缓慢正则表达式匹配的一些情况(Tom)
- 修复通过内存管理引起的 `xml` 处理中各种崩溃(Tom)

有首次应用在8.4中变化的备份补丁。 8.3的代码是有问题的，但新的代码与不想要备份补丁完全不同， 直到它已经获得一些现场测试。

- 修复试图更新复合类型数组列元素的字段中的错误(Tom)
- 当结束位置正好在段边界的时候，修复备份历史文件中 `STOP WAL LOCATION` 项 用来报告下一个WAL段名字(Itagaki Takahiro)
- 修复临时文件泄露的一些情况(Heikki)

  这纠正了之前次要版本中引入的一个问题。 失败的情况是当在另一个函数的异常处理程序中调用plpgsql函数返回集的时候。

- 完善布尔变量情况下约束排除处理， 特别是有可能排除具有"bool_column = false"约束的一个分区(Tom)
- 当读取 `pg_hba.conf` 和相关文件的时候，不要将 `@something` 看作为 文件包含请求，如果 `@` 出现在引号标记内； 另外，也不要把 `@` 本身看作为文件包含请求(Tom)

  如果角色或者数据库名字以 `@` 开头，那么避免不稳定操作。 如果你需要包含一个路径名有空格的文件，那么你仍然可以这样做。但是你必须 书写 `@"/path to/file"` 而不是将引号放在整个结构周围。

- 如果该目录被命名为 `pg_hba.conf` 和相关文件中包含目标，那么 避免一些平台上无限循环(Tom)
- 如果没有设置 `errno` 而 `SSL_read` 或者 `SSL_write` 失败，那么修复可能的无限循环(Tom)

  据说这是openssl的一些Windows版本。

- 不允许本地连接中GSSAPI认证，因为它需要hostname可以正常起作用(Magnus)
- 如果断开连接，那么使用ecpg报告适当的SQLSTATE (Michael)
- 修复psql的 `numericlocale` 选项而不格式化字符串， 它不应该是latex和troff输出格式(Heikki)
- 当指定 `ON_ERROR_STOP` 和 `--single-transaction` 以及隐含 `COMMIT` 期间的错误的时候， 使用psql返回正确退出状态(3) (Bruce)
- 当复合列被设置为空的时候，修复这种情况下plpgsql错误(Tom)
- 当从PL/PerlU中调用PL/Perl函数或者反之亦然，修复可能错误(Tim Bunce)
- 在PL/Python中添加 `volatile` 标记以避免可能指定编译器错误操作(Zdenek Kotala)
- 确保PL/Tcl完全初始化Tcl解析器(Tom)

  如果使用Tcl 8.5或者更高版本，这种疏忽的唯一症状是Tcl `clock` 命令误操作。

- 当许多关键列被指定为`dblink_build_sql_*`函数的时候，避免`contrib/dblink`中崩溃(Rushabh Lathia, Joe Conway)

- 允许`contrib/ltree`操作符中零维数组(Tom)

  这种情况之前作为错误被拒绝，但它更方便地 把它看作同一个零元素数组。 特别是，当`ltree`操作被施加给`ARRAY(SELECT ...)`结果， 并且子选择不会返回行，这避免了不必要的错误。

- 修复通过内存管理引起的`contrib/xml2`中各种崩溃(Tom)

- 使得`contrib/xml2`在Windows上编译更加鲁棒性(Andrew)

- 修复Windows信号处理中竞争条件(Radu Ilie)

  该错误一个显著现象是`pg_listener`中的行在重负载情况下已经被删除。

- 为Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa中DST变化更新时区数据文件到tzdata发布2010e。

E.68. 发布8.3.9

> 发布日期: 2009-12-14

该发布包含来自8.3.8各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.68.1. 迁移到版本8.3.9

运行8.3.X不需要备份/恢复。然而，如果从8.3.8更早版本更新， 参阅8.3.8发布说明。

E.68.2. 变化

- 保护由索引函数导致的间接安全威胁改变会话本地状态(Gurjeet Singh, Tom)

  这个变化阻止不可改变的索引函数可能破坏超级用户会话(CVE-2009-4136)。

- 拒绝在普通名字(CN)字段中包含嵌入空字段的SSL证书(Magnus)

  这可以防止SSL验证期间证书意外匹配到服务器或者客户端名称(CVE-2009-4034)。

- 在后台启动时缓存初始化期间修复可能崩溃(Tom)

- 避免空的同义词词典崩溃(Tom)

- 防止在不安全时期信号中断 `VACUUM` (Alvaro)

  它已经提交元组操作之后，如果 `VACUUM FULL` 被取消，那么该修复程序阻止了PANIC，并且如果有一个普通的 `VACUUM` 在截断该表之后被中断，那么抛出瞬态错误。

- 由于哈希表大小计算中整数溢出，修复可能崩溃(Tom)

  这可能会产生对散列结果大小非常大的规划器估计。

- 修复 `inet` / `cidr` 比较中罕见崩溃(Chris Mikkelson)

- 确保通过预备事务持有的共享元组级别锁被忽略(Heikki)

- 修复用于子事务中被访问的游标的临时文件的过早删除(Heikki)

- 当循环到一个新的CSV日志文件的时候，修复syslogger进程中内存泄露(Tom)

- 修复Windows许可降级逻辑(Jesse Morris)

  这修复了数据库在Windows上启动失败的一些情况，通常带有误导性错误信息比如 "没有找到匹配的postgres可执行文件"。

- 当分裂取决于索引非第一列的时候，修复GiST索引页分裂不正确逻辑(Paul Ramsey)
- 如果在检查点结尾回收利用或者删除旧的WAL文件失败，不要出错误(Heikki)

  更好地将这个问题看作是非致命性的，并且允许检查点来完成。 未来检查点将重新尝试删除。这些问题预计不会在正常操作中， 但都被认为通过错误设计的Windows杀毒和备份软件造成的。

- 确保Windows上WAL文件不要反复被存档(Heikki)

  如果一些其他进程干扰不再需要文件的删除，那么可能发生另外一种现象。

- 修复PAM密码处理更加具有鲁棒性(Tom)

  之前代码伴随Linux `pam_krb5` PAM模块和Microsoft Active Directory结合为 域控制器而失败，它可能在其他地方有问题， 因为它采取不合理的假设关于PAM堆栈可能会 传递给它什么参数。

- 在GSSAPI和SSPI认证方法中提高最大认证令牌（Kerberos标签）大小(Ian Turner)

  当旧的2000字节限制对于Unix Kerberos实现足够的时候， 通过Windows域控制器发出的标签更大。

- 重启序列访问统计收集(Akira Kurosawa)

  这用于执行但在8.3中被损坏。

- 修复 `CREATE OR REPLACE FUNCTION` 期间所有依赖关系的处理(Tom)
- 修复 `WHERE``_x_` = `_x_` 条件的错误处理(Tom)

  在某些情况下这可能作为多余的被忽略，但是它们不是— 它们等同于 `_x_ IS NOT NULL` 。

- 使得文本搜索解析器可以接受XML属性中下划线(Peter)
- 修复 `xml` 二进制输入中编码处理(Heikki)

  如果XML头不指定编码，我们假设缺省UTF-8；之前处理是不一致的。

- 修复从 `plperlu` 调用 `plperl` 错误或者反之亦然(Tom)

  从内部函数的错误退出可能会导致崩溃，由于外部函数未能重新选择正确的Perl解释器。

- 当重新定义PL/Perl函数的时候，修复会话存在期内存泄露(Tom)
- 当通过集合返回PL/Perl函数的时候，确保Perl数组被正确转换为PostgreSQL数组 (Andrew Dunstan, Abhijit Menon-Sen)

  为非设置返回返回正确执行。

- 修复PL/Python中异常处理中罕见崩溃(Peter)
- 在 `contrib/pg_standby` 中，禁用Windows上带有信号的触发器故障转移(Fujii Masao)

  这从来没有任何有用的，因为Windows不具备Unix风格信号，但最近发生的变化使它真正崩溃。

- 确保psql的flex模块与正确的系统头定义一起被编译(Tom)

  这修复了平台上编译错误，该平台上 `--enable-largefile` 导致产生代码中 不兼容变化。

- 使postmaster忽略连接请求数据包中任何 `application_name` 参数， 以提高未来libpq版本的兼容性(Tom)
- 更新时区缩写文件以匹配当前情况(Joachim Wieland)

  这包含添加 `IDT` 和 `SGT` 到缺省时区缩写集合。

- 为了Antarctica, Argentina, Bangladesh, Fiji,Novokuznetsk, Pakistan, Palestine, Samoa, Syria中DST变化更新时区数据文件到tzdata发布2009s； 同时为了Hong Kong历史修正。

E.69. 发布8.3.8

> 发布日期: 2009-09-09

该发布包含来自8.3.7中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.69.1. 迁移到版本8.3.8

运行8.3.X不需要备份/恢复。然而，如果你在 `interval` 列上有 任何哈希索引，你必须在更新到8.3.8之后 `REINDEX` 它们。 另外，如果你从8.3.5更早版本更新，参阅8.3.5发布说明。

E.69.2. 变化

- 修复Windows共享内存分配代码(Tsutomu Yamada, Magnus)

  该错误导致经常报道的"无法重新连接到共享内存"错误信息。

- 在 `pg_start_backup()` 期间强制WAL段切换(Heikki)

  这避免了可能使得基础备份无法使用的一种情况。

- 在安全定义函数中不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` (Tom, Heikki)

  这包含了之前补丁中忽略的一种情况，即在安全定义函数中不允许 `SET ROLE` 和 `SET SESSION AUTHORIZATION` (参阅CVE-2007-6600)

- 使得已加载的加载模块的 `LOAD` 为空操作(Tom)

  之前， `LOAD` 尝试卸载并且重新加载模块，但是这是不安全的并且不是所有都有用。

- 在LDAP身份认证期间不允许空密码(Magnus)

- 修复在外部层聚集函数的参数中的子SELECT处理(Tom)

- 修复从排序或物化规划节点输出中获取整行值相关联的错误(Tom)

- 避免 `synchronize_seqscans` 改变滚动结果和 `WITH HOLD` 游标(Tom)

- 当在AND或者OR列表中有超过100子句的时候， 恢复规划器改变无效部分索引和限制排除优化(Tom)

- 为数据类型 `interval` 修复哈希计算(Tom)

  为区间值上哈希连接纠正错误结果。 这也改变了区间列哈希索引的内容。 如果您有任何这样的索引，你必须更新后 `REINDEX` 它们。

- 将 `to_char(..., 'TH')` 作为带有 `'HH'` / `'HH12'` 后缀的大写字母序数(Heikki)

  它之前作为 `'th'` (小写)被处理。

- 当 `_x_` 超过2百万，并且整数日期在使用中的时候， 修复 `INTERVAL '``_x_` ms'溢出(Alex Hunsaker)

- 修复点和线段之间距离的计算(Tom)

  这导致来自一些几何算子的不正确结果。

- 修复 `money` 数据类型以便在货币量没有小数位数的情况下执行，比如Japan(Itagaki Takahiro)

- 修复模式包含 `%_` 情况下的 `LIKE` (Tom)

- 正确四舍五入日期输入比如 `00:12:57.9999999999999999999999999999` (Tom)

- 修复XML操作中内存泄露(Tom)

- 修复GiST R-tree操作符类中页面分割点的错误选择(Teodor)

- 确保"fast shutdown"请求将强制终止打开会话， 即使"smart shutdown"已经在进行中(Fujii Masao)

- 当输入值（基本）按排序顺序时，避免批量插入到GIN索引中性能下降(Tom)

- 在PL/pgSQL中的某些情况下正确执行NOT NULL域限制(Tom)

- 修复plperl初始化中可移植性问题(Andrew Dunstan)

- 如果 `postgresql.conf` 是空的，修复pg_ctl不进入 无限循环(Jeff Davis)

- 当有许多大对象的时候，提高pg_dump的效率(Tamas Vincze)

- 使用 `SIGUSR1` 而不是 `SIGQUIT` 作为pg_standby故障信号(Heikki)

- 使用pg_standby的 `maxretries` 选项作为记录(Fujii Masao)

- 当键或者值太长而不适合它的数据结构的时候，而不是默默地截断它， 那么 `contrib/hstore` 抛出一个错误。

- 修复 `contrib/xml2` 的 `xslt_process()` 用来正确处理参数的最大数（20）(Tom)

- 提高libpq代码鲁棒性以恢复 `COPY FROM STDIN` 中错误(Tom)

- 当安装两个库的时候，避免包含readline和editline头文件冲突(Zdenek Kotala)

- 为Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan（历史修正） Mauritius, Morocco, Palestine, Syria, Tunisia 中DST变化更新时区数据文件到tzdata 发布2009l。

E.70. 发布8.3.7

> 发布日期: 2009-03-16

该发布包含来自8.3.6中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.70.1. 迁移到版本8.3.7

运行8.3.X不需要备份/恢复。然而，如果从8.3.5更早版本更新，参阅8.3.5发布说明。

E.70.2. 变化

- 当编码转换失败的时候，避免错误递归崩溃(Tom)

  该变化为相关错误情况在最后两个次要版本中扩展修复。 之前修复程序为最初问题报告进行了细化， 但我们现在已经认识到通过编码转换函数抛出的任何错误 可能潜在地导致无限递归，而试图报告错误。 如果我们发现已经卷入了一个递归错误报告的情况时， 解决的办法是禁用转换以及编码转换并报告任何纯ASCII格式错误消息。

- 不允许为指定转换函数带有错误编码的 `CREATE CONVERSION` (Heikki)

  这可以防止编码转换失败的情况。之前变化是预防同一区域其它类型错误的手段。

- 修复 `xpath()` 不会修改路径表达式除非必要，并且当必要时做出理智尝试(Andrew)

  SQL标准表明 `xpath` 致力于数据是一个文档片段， 但libxml不支持这一点， 其实这是不明确的，按照XPath标准是明智的。 `xpath` 试图通过修改数据和路径表达式解决这个错误匹配， 但是修改是古怪的，并可能导致有效的搜索失败。现在， `xpath` 检查数据是否实际上是一个良好的文档， 并且如果是这样调用不改变数据或路径表达式的libxml。 否则，一个不太可能失败的不同修改方法被使用。

  > **Note:** 新的修改方法仍然不是100%令人满意，并且它 似乎没有真正的解决方案是可能的。 这个补丁因此被看作是一个短期有效的防止不必要的破坏现有应用程序。 PostgreSQL 8.4直接拒绝在不是一个良好文档的数据上使用 `xpath` 。

- 当 `to_char()` 指定格式代码对于数据参数类型不合适的时候，修复核心转储(Tom)

- 当C语言环境用于多字节编码的时候，修复文本搜索中可能失败(Teodor)

  在平台上有可能崩溃，即 `wchar_t` 比 `int` 更窄的时候，尤其Windows上。

- 修复文本搜索解析器的处理类似电子邮件包含多个 `@` 字符串效率低下的情况(Heikki)

- 修复较大子查询输出列表中子 `SELECT` 规划器问题(Tom)

  这个错误已知现象是"未能定位分组列"依赖于涉及的数据类型错误； 但是也有可能是其它问题。

- 修复隐式强制 `CASE WHEN` 的反编译(Tom)

  当尝试检查或者备份视图的时候，这个错误可能导致断言启动编译中断言错误，或者是其它情况中 "意外的CASE WHEN 子句"错误消息。

- 修复TOAST表的行类型拥有者可能错误分配(Tom)

  如果 `CLUSTER` 或者 `ALTER TABLE` 的重写形式通过某人而不是表的所有者被执行， `pg_type` 项为表的TOAST表将最终标记为由别人所拥有。 这没有造成直接的问题， 因为普通的数据库操作不会检查TOAST rowtype的权限。 然而，它可能会导致意外故障，如果之后尝试删除发出该命令的角色（在8.1或者8.2中）， 或者已经这样做之后（在8.3中）来自pg_dump中的"数据类型所有者似乎无效"警告。

- 如果当前会话从来没有执行任何 `LISTEN` 命令，那么改变 `UNLISTEN` 迅速退出(Tom)

  多数情况下这不是一个特别有用的优化， 但因为 `DISCARD ALL` 调用 `UNLISTEN` ， 之前编码导致为大量使用 `DISCARD ALL` 的应用程序带来巨大的性能问题。

- 修复PL/pgSQL没有把 `INTO` 在 `INSERT` 之后看作字符串任意位置的一个INTO变量子句，不仅在开始；尤其是，不会在 `CREATE RULE` 中 `INSERT INTO` 中失败(Tom)

- 在块退出时完全清理PL/pgSQL错误状态变量(Ashesh Vashi和Dave Page)

  这不是PL/pgSQL本身存在的问题，但当检查一个函数的状态的时候，该疏忽可能导致PL/pgSQL调试器崩溃。

- 在Windows上重新尝试失败调用到 `CallNamedPipe()` (Steve Marshall, Magnus)

  看起来这个函数有时可能会暂时失效; 我们之前将任何故障看作是严重的错误， 这可能混淆 `LISTEN` / `NOTIFY` 以及其它操作。

- 添加 `MUST` (Mauritius Island Summer Time)到已知的时区缩写缺省列表中(Xavier Bugaud)

E.71. 发布8.3.6

> 发布日期: 2009-02-02

该发布包含来自8.3.5的各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.71.1. 迁移到版本8.3.6

运行8.3.X不需要备份/恢复。然而，如果从8.3.5更早版本更新，参阅8.3.5发布说明。

E.71.2. 变化

- 使用 `DISCARD ALL` 发布咨询锁，除了它已经做的之外(Tom)

  这是最适合的操作。然而，这可能影响现有的应用程序。

- 修复整个索引GiST扫描以便正确执行(Teodor)

  如果在GiST索引上集群一个表，那么这个错误可能导致行丢失。

- 修复 `xmlconcat(NULL)` 的崩溃(Peter)
- 如果高比特位字符被用作标志，那么修复 `ispell` 词典中可能崩溃(Teodor)

  这被称为是由一个广泛可用的Norwegian字典完成的， 并且在其它中相同的条件可能存在。

- 修复为复合类型pg_dump错误顺序输出(Tom)

  最可能的问题是在索引和视图需要它们之后为用户定义的运算符类而被备份。

- 完善 `headline()` 函数中URL的处理(Teodor)
- 完善 `headline()` 函数中超长headline的处理(Teodor)
- 如果用错误的转换函数为指定编码对创建一个编码转换， 防止可能的断言故障或错误转换。(Tom, Heikki)
- 修复可能的断言失败，如果在PL/pgSQL中执行语句改写成另一种语句， 例如，如果 `INSERT` 改写为 `UPDATE` (Heikki)
- 确保快照可用于数据类型输入函数(Tom)

  这主要影响了声明为涉及用户定义的稳定或者不变函数 `CHECK` 限制的域。如果没有设置快照，则这样的函数往往失败。

- 为用于数据类型I/O中SPI使用函数使其更安全；特别是用于域检查约束(Tom)
- 避免 `VACUUM` 中小表的不必要锁定(Heikki)
- 修复保持 `ALTER TABLE ENABLE/DISABLE RULE` 被活跃会话识别的问题(Tom)
- 修复使用 `UPDATE RETURNING tableoid` 返回零而不是正确OID问题(Tom)
- 允许函数声明为采用 `ANYARRAY` 以便在该类型的 `pg_statistic` 列上执行(Tom)

  这曾经运行，但是在8.3中无意被损坏。

- 当转变的平等应用于外部连接子句的时候，修复选择性的规划器误评估(Tom)

  这可能会导致查询不好的规划像 `... from a left join b on a.a1 = b.b1 where a.a1 = 42 ...`。

- 完善长 `IN` 列表的优化处理(Tom)

  当启用约束排除的时候，该变化避免了在列表上浪费大量时间。

- 防止在GIN索引构建期间同步扫描(Tom)

  由于GIN是在增加TID顺序中为插入元组进行了优化， 选择使用同步扫描可能会减缓通过三个或更多个因素的编译。

- 确保持有游标的内容不依赖于TOAST表的内容(Tom)

  此前，游标中大字段值可能会表示为TOAST指针，如果引用的表在读取游标之前被删除， 或者如果大值被删除，然后清理。这可能失败。这不能发生在普通游标中， 但它可能使用游标保持超过它创建的事务。

- 当一组返回函数没有读取整个结果而被终止，那么修复内存泄露(Tom)
- 当数据库编码不是UTF-8的时候，修复XML函数中编码转换问题(Tom)
- 修复 `contrib/dblink` 的 `dblink_get_result(text,bool)` 函数(Joe)
- 修复来自 `contrib/sslinfo` 函数的垃圾输出(Tom)
- 当在命令中被触发一次以上的时候，修复 `contrib/tsearch2` 兼容性触发器的 不正确操作(Teodor)
- 修复autovacuum中可能错误信号(Heikki)
- 支持作为Windows 7 beta服务运行(Dave和Magnus)
- 修复varchar结构ecpg的处理(Michael)
- 当无法获得PL/Perl连接信息的时候，修复configure脚本用来正确报告 错误(Andrew)

- 使用适当的所有文档引用`pgsql-bugs`和/或者`pgsql-hackers`， 而不是现在过时的`pgsql-ports`和`pgsql-patches`邮件列表(Tom)
- （为Kathmandu和Switzerland, Cuba中历史DST）更新时区数据文件到tzdata发布2009a

E.72. 发布8.3.5

> 发布日期: 2008-11-03

该发布包含来自8.3.4各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.72.1. 迁移到版本8.3.5

运行8.3.X不需要备份/恢复。然而，如果从8.3.1更早版本更新，参阅8.3.1发布说明。 另外，如果你正在运行之前的8.3.X版本，推荐更新之后 `REINDEX` 所有 GiST索引。

E.72.2. 变化

- 修复GiST索引损坏，由于在删除之后标记错误索引项"死的"(Teodor)

  这可能导致索引搜索无法找到他们应该找的行。 损坏的索引可以使用 `REINDEX` 被修复。

- 当客户端编码不能表示本地化错误消息的时候，修复后台崩溃(Tom)

  我们已经解决了之前类似问题，如果"字符没有等同"消息自身不能被转化， 但是可能仍然失败。当我们发现这种情况的时候，解决办法是禁用本地化并且发送 纯ASCII错误消息。

- 修复 `bytea` 到XML映射中可能崩溃(Michael McMaster)

- 当深度嵌套函数被触发器调用的时候，修复可能崩溃(Tom)

- 提高 `_expression_` `IN` (`_expression-list_`)查询优化(Tom,每一个想法都来自Robert Haas)

  右边有查询变量比起之前版本已经在8.2.x和8.3.x中被低效处理， 该修复为这种情况修复了8.1操作。

- 当子 `SELECT` 出现在 `FROM` ， 多行 `VALUES` 列表，或者 `RETURNING` 列表中的函数调用中的时候，修复规则查询的错误扩展(Tom)

  这个问题的常见现象是"未知节点类型"错误。

- 在GiST索引的 `IS NULL` 搜索的重复扫描中，修复断言失败(Teodor)

- 在散列聚集规划的重复扫描中，修复内存泄露(Neil)

- 当新定义PL/pgSQL触发器函数作为正常函数被调用的时候， 确保报告一个错误(Tom)

- 在 `CREATE DATABASE` 开始拷贝文件之前，强迫检查点(Heikki)

  如果在源数据库中已经删除了文件，这可以防止可能错误。

- 当移动表到另外一个使用 `ALTER SET TABLESPACE` 的表空间时，预防 `relfilenode` 数的可能冲突(Heikki)

  该命令尝试重新使用已有文件名，而不是选择在目标目录中未使用的一个。

- 当单个查询项匹配文本第一个字的时候，修复不正确文本搜索headline生成(Sushant Sinha)

- 当在 `--enable-integer-datetimes` 编译中使用非ISO日期类型时， 修复间隔值中小数秒不正确显示(Ron Mayer)

- 当他们被逃逸的时候，使用 `ILIKE` 比较字符不区分大小写(Andrew)

- 确保 `DISCARD` 通过语句记录被正确处理(Tom)

- 在PITR恢复期间修复最后完成事务时不正确记录(Tom)

- 当传递的元组以及元组描述符有不同列数时，确保 `SPI_getvalue` 和 `SPI_getbinval` 正确操作(Tom)

  当表中有列被添加或者删除的时候，这种情况是正常的， 但是这两个函数不能正确处理它。唯一可能后果是不正确错误显示。

- 标记 `SessionReplicationRole` 为 `PGDLLIMPORT` ， 因此它可以用于Windows上Slony (Magnus)

- 当使用libpq的 `gsslib` 参数的时候， 修复小的内存泄露(Magnus)

  在连接关闭时不释放通过参数字符串使用的空格。

- 如果需要，那么确保libgssapi被连接到libpq中(Markus Schaaf)

- 修复 `CREATE ROLE` 的ecpg的解析(Michael)

- 修复 `pg_ctl restart` 最近破损(Tom)

- 确保以二进制模式打开 `pg_control` (Itagaki Takahiro)

  pg_controldata和pg_resetxlog这样做不正确， 因此在Windows上可能失败。

- 更新时区数据文件到tzdata发布2008i (为Argentina, Brazil, Mauritius, Syria中DST变化)

E.73. 发布8.3.4

> 发布日期: 2008-09-22

该发布包含来自8.3.3中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.73.1. 迁移到版本8.3.4

运行8.3.X不需要备份/恢复。然而，如果从8.3.1更早版本更新，参阅8.3.1发布说明。

E.73.2. 变化

- 修复btree WAL恢复编码中错误(Heikki)

  如果通过页拆分操作中途结束WAL，那么恢复失败。

- 修复为HOT页修剪错误截断XID的潜在使用(Alvaro)

  这个错误产生通过 `VACUUM` 查阅的系统目录损坏风险： 死的元组版本可能太早被删除。在实际数据库操作中的影响可能最小， 因为当检查目录的时候，该系统并不遵循MVCC规则， 但它可能会从pg_dump或其它客户端程序输出中导致瞬时错误。

- 修复 `datfrozenxid` 的潜在错误计算(Alvaro)

  这个错误可能解释了删除旧的 `pg_clog` 数据一些错误报告。

- 在 `pg_class` 重新被索引之后修复不正确的HOT更新(Tom)

  如果在同一会话中 `REINDEX TABLE pg_class` 伴随 `ALTER TABLE RENAME` 或者 `ALTER TABLE SET SCHEMA` 命令，可能发生 `pg_class` 损坏。

- 修复丢失的"combo cid"情况(Karl Schnaitter)

  这个错误使得行对于通过所有终止的多个子事务已经删除的事务不可见。

- 如果该表目前的检查在错误的时间被删除， 那么避免自动清理崩溃(Alvaro)

- 从32位到64位扩展本地锁计数器(Tom)

  该反馈是计数器可能在相当长事务中溢出，导致意想不到的"持有锁"错误。

- 修复GiST索引扫描期间元组的重复输出(Teodor)

- 当修改任何一个表的时候，从头回收外键检查查询(Tom)

之前，8.3可能尝试重新规划查询，但是可能操作之前生产的查询文本。 如果重命名表或者列，那么导致错误。

- 当视图包含一个简单的 `UNION ALL` 结构的时候， 修复丢失的权限检查(Heikki)

  正确检查引用表权限，但不是视图本身权限。

- 在执行器启动中添加检查以确保通过 `INSERT` 或者 `UPDATE` 产生的元组匹配目标表的当前行类型(Tom)

  这种情况在8.3中是不可能的，但是在以前版本中可以发生，所以检查似乎是谨慎的。

- 修复 `DROP OWNED` 期间可能的重复删除(Tom)

  这通常会导致奇怪错误比如"对于关系NNN缓存查找失败"。

- 修复XML操作中若干内存泄露(Kris Jurka, Tom)
- 为不可接受的目标数据类型修复 `xmlserialize()` 正确改善误差(Tom)
- 修复文本搜索配置文件解析中错误处理多字节字符的地方(Tom)

  配置文件中出现的某些字符总是引起"无效字节序列编码"错误。

- 为文本搜索配置文件中报道的所有错误提供文件名和行号位置(Tom)
- 修复 `AT TIME ZONE` 首先尝试解析它的时区参数为时区缩写， 并且如果失败，则尝试作为完整时区名，而不是之前的其它方式(Tom)

  时间戳输入函数一直按此顺序解决模棱两可的区域名称。 采用 `AT TIME ZONE` 这样做提高了一致性，并且修复了8.1引入的一个兼容性错误： 在模棱两可的情况下，我们现在操作和8.0以及以前的操作都是一样的， 因为在旧版本中 `AT TIME ZONE` 接受*only*缩写。

- 当在64位平台上运行的时候，修复datetime输入函数以正确检测整数溢出(Tom)
- 当显示有单位的配置参数的时候，防止单位换算期间整数溢出(Tom)
- 改善写很长日志信息到syslog的性能(Tom)
- 允许 `pg_hba.conf` 中LDAP URL的后缀部分空格(Tom)
- 修复 `SELECT DISTINCT ON` 查询上游标向后扫描中的错误(Tom)
- 修复规划器错误可能不正确地推翻外部连接下面的 `IS NULL` 测试(Tom)

  在大写 `OR` 子句的基础上通过 `IS NULL` 测试同一关系触发。

- 修复嵌套子select表达式规划器错误(Tom)

  如果外部子select对父查询没有直接相关性，但内部确实如此， 可能不会为新的父查询行计算外部值。

- 修复规划器以估计产生布尔结果的 `GROUP BY` 表达式总是产生两组， 不管表达式的内容(Tom)

  比起规则 `GROUP BY` 评估某个布尔测试像 `_col_``IS NULL` 来说 更加准确。

- 当 `FOR` 循环的目标变量是包含复合类型字段的记录，修复 PL/pgSQL而不失败(Tom)
- 修复PL/Tcl以正确操作Tcl 8.5， 并且更加小心数据编码发送的或者来自Tcl的(Tom)
- 提高 `PQescapeBytea()` 的性能(Rudolf Leitgeb)
- 在Windows上，通过避免libpq尝试发送超过64KB每个系统调用解决Microsoft错误(Magnus)
- 修复ecpg正确处理 `SET` 命令中变量(Michael)
- 在错误发送一个SQL命令之后完善pg_dump和pg_restore的错误报告(Tom)
- 修复pg_ctl以正确保存通过 `restart` 的postmaster命令行参数(Bruce)
- 修复pg_standby中错误的WAL文件截止点计算(Simon)
- （为 Argentina, Bahamas, Brazil, Mauritius, Morocco,Pakistan, Palestine和Paraguay中DST变化） 更新时区数据文件到 tzdata发布2008f。

E.74. 发布8.3.3

> 发布日期: 2008-06-12

该发布中包含8.3.2中修复的一个严重的和较小的错误。 关于8.3主要发布中新特性信息，参阅Section E.77。

E.74.1. 迁移到版本8.3.3

运行8.3.X不需要备份/恢复。然而，如果从8.3.1更早版本更新，参阅8.3.1发布说明。

E.74.2. 变化

- 采用 `pg_get_ruledef()` 括起负的常量(Tom)

  该修复程序之前，视图或规则中负常数可能被备份， 比方说， `-42::integer` ， 这是不正确的： 它应该是 `(-42)::integer` ， 由于运算符优先级规则。 通常这样差别不大， 但它可能与其它最近补丁相互作用导致PostgreSQL拒绝 有效的 `SELECT DISTINCT` 视图查询内容。 因为这可能会导致pg_dump输出未能重新加载，它被视为一个高优先级补丁。 转储输出实际上是不正确的唯一的发行版本是8.3.1和8.2.7。

- 使用 `ALTER AGGREGATE ... OWNER TO` 更新 `pg_shdepend` (Tom)

  如果之后在 `DROP OWNED` 或者 `REASSIGN OWNED` 操作中涉及到聚集，那么 这一疏忽可能导致错误。

E.75. 发布8.3.2

> 发布日期: 从未公布

该发布中包含8.3.1中各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.75.1. 迁移到版本8.3.2

运行8.3.X不需要备份/恢复。然而，如果从8.3.1更早版本更新，参阅 8.3.1发布说明。

E.75.2. 变化

- 当使用UTF-8数据库编码和不同客户端编码的时候，修复在Windows上发生的 `ERRORDATA_STACK_SIZE exceeded` 崩溃(Tom)
- 为 `recovery_command` 参数中 `%r` 宏修复不正确 归档截断点计算(Simon)

  如果热备份脚本依赖于 `%r` 决定何时丢弃WAL分段文件， 这可能导致数据丢失。
- 修复 `ALTER TABLE ADD COLUMN ... PRIMARY KEY` ，使得新列被正确检查以查看 它是否被初始化为所有非空(Brendan Jurd)

  之前版本忽略检查这项需求。
- 修复 `REASSIGN OWNED` 以致于工作于程序语言(Alvaro)
- 修复在非 `SELECT` 顶级操作查询中 `SELECT FOR UPDATE/SHARE` 作为子查询出现的问题(Tom)
- 当从同一个祖先中继承约束的多个父关系中继承"同一"约束的时候， 修复可能的 `CREATE TABLE` 错误(Tom)
- 修复 `pg_get_ruledef()` 以显示别名， 如果存在，附加到 `UPDATE` 或者 `DELETE` 的目标表中(Tom)
- 恢复之前8.3操作TID用在一个TidScan规划结果中静默地不匹配任何行中的范围外块号(Tom)

  8.3.0和8.3.1相反抛出一个错误。
- 修复可能导致 `许多LWLocks` 失败的GIN错误(Teodor)
- 为 `tsquery` 修复受损GiST比较函数(Teodor)
- 修复 `tsvector_update_trigger()` 和 `ts_stat()` 用来接受超出它们期望类型域(Tom)

- 修复故障以支持枚举数据类型作为外键(Tom)
- 当解压损坏的数据时，避免可能崩溃(Zdenek Kotala)
- 修复在延迟断开和 `DROP DATABASE` 之间竞态条件(Heikki)

  在最坏的情况下，这可能会导致在一个新的数据库删除新创建的表中， 获得相同OID作为最近删除的; 但当然这是非常小概率情况。

- 修复后端的SIGTERM退出可能遗留在共享内存中损坏状态中的两个地方(Tom)

  两种情况都不是很重要，如果SIGTERM用于关闭整个数据库集群， 但是如果尝试SIGTERM个人后端，可能有问题。

- 修复可能的崩溃，由于当 `_x_` 和 `_y_` 具有不同的数据类型的时候， 不正确规划引起 `_x_` IN (SELECT `_y_` FROM ...)子句。 当从 `_y_` 的类型转换为 `_x_` 的类型有损耗时， 确保该操作是语义正确的。
- 修复疏忽，避免规划器替代已知的Param值好像它们是常数(Tom)

  这个错误部分禁用8.3.0和8.3.1中未命名扩展查询语句优化： 特别是如果LIKE模式作为参数被传递，那么LIKE到索引扫描优化将永远不会被应用， 并且取决于参数值的约束排除也不能正常运行。

- 当可索引的 `MIN` 或者 `MAX` 聚集用于 `DISTINCT` 或者 `ORDER BY` 的时候，修复规划器错误(Tom)
- 修复规划器以确保它为了正提供排序节点的规划节点从来不使用"physical tlist"(Tom)

  这导致排序摆布比实际需要的更多数据， 因为未使用列值被包含在排序数据中。

- 避免查询字符串的不必要拷贝(Tom)

  当许多命令作为单一查询字符串被提交的时候，修复了8.3.0中介绍的性能问题。

- 当检查子事务XID的时候，使得 `TransactionIdIsCurrentTransactionId()` 使用 二进制搜索而不是线性搜索(Heikki)

  这修复了8.3.0中比之前版本显著缓慢的情况。

- 修复ISO-8859-5和其它编码之间转换用来处理Cyrillic "Yo"字符 (使用两个点的 `e` 和 `E`) (Sergey Burladyan)
- 修复一些数据类型输入函数，尤其是 `array_in()` ，被允许结果中未使用字节包含 未初始化，不可预测的值(Tom)

  这可能导致错误，其中两个似乎相同文本值被认为不相同，导致解析器抗议 不匹配的 `ORDER BY` 和 `DISTINCT` 表达式。

- 修复正则表达式子字符串匹配的情况(`substring(``_string_` 来自 `_pattern_`)) (Tom)

  当有一个匹配模式整体，但用户指定括号子表达式， 并且子表达式还没有得到匹配。那么出现问题， 一个例子是 `substring('foo' from 'foo(bar)?')` 。 这应该返回NULL，因为 `(bar)` 不匹配，但 它错误地返回全模式匹配（即 `foo` ）。

- 阻止发动自动清理防止XID重叠的取消(Alvaro)
- 完善未确定元组的 `ANALYZE` 的处理（通过尚未提交的事务插入或者删除），以使 它报告给统计收集器的数量可能是正确的(Pavan Deolasee)
- 修复initdb拒绝 `--xlogdir` (`-X`)选项的相对路径(Tom)
- 采用psql输出标签字符作为适当空间数， 而不是8.3.0和8.3.1中执行的 `\x09` (Bruce)
- 更新时区数据文件到tzdata发布2008c (为Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba和Argentina/San_Luis中DST变化)
- 添加 `ECPGget_PGconn()` 函数到ecpglib (Michael)
- 修复来自ecpg的 `PGTYPEStimestamp_sub()` 函数不正确结果(Michael)
- 修复ecpg中连续行标记的处理(Michael)
- 修复 `contrib/cube` 函数中可能的崩溃(Tom)
- 当输入查询返回NULL值的时候，修复 `contrib/xml2` 的 `xpath_table()` 函数中核心转储(Tom)
- 修复 `contrib/xml2` 的makefile而不覆盖 `CFLAGS` , 并且为libxslt存在或者不存在使其自动配置(Tom)

E.76. 发布8.3.1

> 发布日期: 2008-03-17

该发布包含来自8.3.0的各种修复。关于8.3主要发布中新特性信息， 参阅Section E.77。

E.76.1. 迁移到版本8.3.1

运行8.3.X不需要备份/恢复。然而，如果你受到下面描述的Windows区域问题的影响。 在更新之后你可能需要文本列上 `REINDEX` 索引。

E.76.2. 变化

- 修复认为不同字符组合是相同的Windows区域的字符串比较(Tom)

  当使用UTF-8数据库编码的时候，此修复程序仅适用于Windows。 相同的修复程序解决了2年前所有其他情况，但是使用UTF-8的Windows使用 未被更新的单独代码路径。如果你正使用认为非相同字符串是相同的区域， 你可能需要 `REINDEX` 以修复文本列上现有索引。

- 修复 `VACUUM FULL` 中极端情况错误(Tom)

  8.2中介绍了不同的系统目录中并发 `VACUUM FULL` 操作之间的 潜在死锁。这已得到纠正。8.3执行的更糟糕， 因为死锁可能出现在关键代码部分，使其PANIC而不仅仅是ERROR条件。

  此外， `VACUUM FULL` 通过清理系统目录中途失败可能会导致并发数据库会话中缓存损坏。

  当处理没有活元组页的时候， 8.3中介绍的另外一个 `VACUUM FULL` 可能导致崩溃或者内存不足报告。

- 修复涉及 `character` 或者 `bit` 列的外键检查错误操作(Tom)

  如果引用列不同除兼容类型（比如 `varchar` ），那么错误强制约束。

- 在无操作外键检查中避免不必要死锁错误(Stephan Szabo, Tom)

- 当重新规划预备查询的时候，修复可能的核心转储(Tom)

  该错误只影响协议级别预备操作，而不是SQL `PREPARE` ， 因此常常被认为JDBC, DBI以及大量使用预备语句的其它客户端驱动程序。

- 当重新规划调用SPI使用函数的查询时，修复可能错误(Tom)

- 修复逐行比较涉及不同数据类型列中的错误(Tom)
- 修复长期存在的 `LISTEN` / `NOTIFY` 竞态条件(Tom)

  在极少数情况中执行 `LISTEN` 的会话中可能无法得到通知，即使可以被预计，因为 执行 `NOTIFY` 的并发事务被观察后提交。

  该修复的负面效果是已经执行尚未提交的 `LISTEN` 命令的事务 将不能看到任何 `pg_listener` 中 `LISTEN` 行，应该选择查看；之前可能会。这种操作从来没有记录一种方式或者其它， 但是可能某些应用程序依赖于旧操作。

- 在一个预备事务中不允许 `LISTEN` 和 `UNLISTEN` (Tom)

  之前这是被允许的，但是尝试执行它有各种不好的结果，尤其是原始后台不能退出 只要 `UNLISTEN` 没有提交。

- 不允许删除预备事务中临时表(Heikki)

  8.1中不允许，但是在8.2和8.3中无意中损坏了该检查。

- 当在使用哈希索引的查询中发生错误的时候，修复罕见崩溃(Heikki)
- 修复 `tsquery` 值的不正确比较(Teodor)
- 修复单字节编码中非ASCII字符 `LIKE` 不正确操作(Rolf Jentsch)
- 禁用 `xmlvalidate` (Tom)

  该函数应该在8.3版本之前删除，但是无意中留在源代码中。它造成小的安全风险， 因为未授权用户可以使用它读取访问服务器任何文件的前几个字符。

- 修复集合返回函数的某些用法的内存泄露(Neil)
- 使用 `encode(` *bytea* `, 'escape')` 转换所有 高位字节值到 `\``_nnn_` 八进制转义序列(Tom)

  当数据库编码是多字节时，有必要避免编码问题。 这种变化可能为预期从 `encode` 中指定结果的应用中产生兼容问题。

- 修复公元前2月29号日期时间值的输入(Tom)

  关于哪一年是闰年前者编码是错误的。

- 修复 `ALTER OWNER` 的一些变量中"未知节点类型"错误(Tom)
- 避免 `CREATE TABLE LIKE INCLUDING INDEXES` 中表空间权限错误(Tom)
- 当中断锁等待的时候，确保 `pg_stat_activity` . `waiting` 标志被清除(Tom)
- 修复Windows Vista上进程权限的处理(Dave, Magnus)

  特别是，这个修复允许作为管理员用户启动服务器。

- 更新时区数据文件到tzdata发布2008a（尤其是，最近Chile变化）；调整时区缩写`VET` (Venezuela)意味着UTC-4:30, not UTC-4:00 (Tom)
- 修复数组ecpg问题(Michael)
- 修复pg_ctl以正确从命令行选项中提取postmaster的端口号(Itagaki Takahiro, Tom)

  之前，`pg_ctl start -w`可能尝试联系错误端口上postmaster，导致启动错误的虚假报告。
- 使用`-fwrapv`防御最近gcc版本中可能的错误优化(Tom)

  当使用gcc 4.3或者更高版本编译PostgreSQL的时候，这是必要的。
- 启动使用MSVC 编译`contrib/uuid-ossp`(Hiroshi Saito)

E.77. 发布8.3

> 发布日期: 2008-02-04

E.77.1. 概述

随着显著的新功能和性能增强，此版本对于PostgreSQL是一个重大的飞跃。 通过越来越多的社区已经大大加快了发展的步伐成为可能。此版本增加了以下主要特性：

- 全文搜索被集合到核心数据库系统中
- 支持SQL/XML标准，包含新的操作以及 `XML` 数据类型
- 枚举数据类型(`ENUM`)
- 复合类型数组
- 通用唯一标识符(`UUID`)数据类型
- 添加控制是否 `NULL` 首先排序还是最后排序
- 可更新游标
- 服务器配置参数可以基于每个函数被设置
- 用户定义类型可以有类型修饰符
- 当更新表定义变化或者统计的时候，自动重新规划缓存查询
- 记录和统计收集中大量优化
- 为Windows上认证支持安全服务提供者接口(SSPI)
- 支持多个并发autovacuum进程，和其它autovacuum优化
- 允许使用Microsoft Visual C++编译整个PostgreSQL发布

主要性能改进如下。大部分这些功能是自动的，不需要用户改变或者调整：

- 在事务提交期间异步提交延迟写入WAL
- 检查点写操作被分散在一个较长时间周期内用来平滑每个检查点中I/O剧增
- 堆元组(HOT)加快了大部分 `UPDATE` 和 `DELETE` 的 空间复用
- 及时后端写策略提高了磁盘写效率
- 为只读事务使用非持久事务ID减少开销和 `VACUUM` 需求

- 减少每个字段和每行存储开销
- 大顺序扫描不再频繁删除已使用缓存页
- 并发大顺序扫描可以共享磁盘读取
- `ORDER BY ... LIMIT` 无需排序

在下面章节更详细解释了上面每一项。

E.77.2. 迁移到版本8.3

使用pg_dump的备份/恢复为了那些希望从任何其它版本 迁移数据。

观察下面的不兼容性：

E.77.2.1. 普遍的

- 非字符数据类型不再自动转换为 `TEXT` (Peter, Tom)

  之前，如果一个非字符值被提供给要求 `text` 输入的操作者或函数， 它会自动转换为 `text` ，针对大多数（但不是全部）内置的数据类型。 这不再发生：为了所有非字符串类型需要显式转换为 `text` 。例如，这些 表达式曾执行：

  ```
  substr(current_date, 1, 4)
  23 LIKE '2%'
  ```

  但是现在分别导致 "函数不存在"和"操作符不存在"错误。 使用显式转换而不是：

  ```
  substr(current_date::text, 1, 4)
  23::text LIKE '2%'
  ```

  （当然，你也可以使用更详细的 `CAST()` 语法了。） 该变化的原因是这些自动强制转换过于频繁引起令人惊讶操作。 一个例子是在以前的版本中，接受这个表达式，但并没有执行预期的内容：

  ```
  current_date &lt; 2017-11-17
  ```

  这实际上比较日期为整数，这应该被（现在是）拒绝— 但双方自动转换都被转换为 `text` 并且执行文本比较， 因为当没有其他的 `<` 操作符的时候， `text < text` 操作符可以匹配表达式。

  类型 char(``_n_ )和 varchar(``_n_)仍然自动转换为 `text` 。 另外，自动转换到 `text` 仍然为输入级联(`||`)操作符执行， 只要至少一个输入是字符串类型。

- 来自 `contrib/tsearch2` 全文搜索特性已经被移动到核心服务器中，有一些小的语法变化

  `contrib/tsearch2` 包含一个兼容接口。

- `ARRAY(SELECT ...)`，其中 `SELECT` 没有返回行，现在返回 一个空数组，而不是 NULL(Tom)

- 基本数据类型的数组类型名不再是带有下划线前缀的基础类型名

  当有可能时，旧的命名约定仍然采用， 但应用程序代码不应该再依赖于它。 代替使用新的 `pg_type.typarray` 列， 以确定与给定的类型相关联的数组数据类型。

- `ORDER BY ... USING` `_operator_` 必须使用小于或者大于在btree操作符类中 定义的 `_operator_`

  为防止不一致结果添加该限制。

- `SET LOCAL` 变化持续直到最外层事务截止，除非回滚(Tom)

  之前在子事务提交后（`RELEASE SAVEPOINT` 或者从PL/pgSQL异常块中退出） 丢失 `SET LOCAL` 的影响

- 在事务块中拒绝的命令在多语句查询字符串中被拒绝(Tom)

  比如， `"BEGIN; DROP DATABASE; COMMIT"` 被拒绝即使作为单一查询消息被提交。

- 事务块外部 `ROLLBACK` 发出 `NOTICE` 而不是 `WARNING` (Bruce)

- 阻止 `NOTIFY` / `LISTEN` / `UNLISTEN` 接受 模式限定名称(Bruce)

  之前，这些命令接受 `schema.relation` 但是忽略模式部分，这是令人困惑的。

- `ALTER SEQUENCE` 不再影响序列的 `currval()` 状态(Tom)

- 外键必须匹配交叉数据类型引用的可索引条件(Tom)

  这改善了语义一致性并且可以避免性能问题。

- 限制对象大小函数到拥有合理权限可以查看该信息的用户(Tom)

  比如， `pg_database_size()` 需要 `CONNECT` 权限， 这缺省授权给每一个人。 `pg_tablespace_size()` 在表空间上需要 `CREATE` 权限，如果表空间是数据库缺省表空间，则允许。

- 删除非法的 `!!=` (非in)操作符(Tom)

  `NOT IN (SELECT ...)` 是执行该操作的正确方式。

- 内部哈希函数更加一致分布(Tom)

如果应用程序代码正使用内部PostgreSQL哈希函数计算并且存储哈希值， 那么哈希值必须被再生。

- 改变处理可变长度数据值C代码约定(Greg Stark, Tom)

  新的 `SET_VARSIZE()` 宏必须用于设置生成 `varlena` 值的长度。 另外，它在更多情况中有必要扩展("de-TOAST")的输入值。

- 连续归档不再报告每个成功归档操作到服务器日志，除非使用 `DEBUG` 级别(Simon)

E.77.2.2. 配置参数

- 管理服务器参数中许多变化

  删除 `bgwriter_lru_percent` ， `bgwriter_all_percent` ， `bgwriter_all_maxpages` ， `stats_start_collector` 和 `stats_reset_on_server_start` 。 `redirect_stderr` 重命名为 `logging_collector` 。 `stats_command_string` 重命名为 `track_activities` 。 `stats_block_level` 和 `stats_row_level` 被合并为 `track_counts` 。 一个新的布尔配置参数 `archive_mode` 控制归档。Autovacuum的缺省设置已改变。

- 删除 `stats_start_collector` 参数(Tom)

  我们现在总是启动收集器处理过程，除非UDP套接创建失败。

- 删除 `stats_reset_on_server_start` 参数(Tom)

  这被删除，因为 `pg_stat_reset()` 用于这一目的。

- `postgresql.conf` 中注释参数导致它恢复到缺省值(Joachim Wieland)

  之前，注释项留下参数的值未改变直到下次服务器重新启动。

E.77.2.3. 字符编码

- 添加对无效编码数据的更多检查(Andrew)

  该变化插入反斜杠逃逸字符串处理和 `COPY` 逃逸处理中存在的一些漏洞。 目前检查脱转义字符串看是否其结果创建了一个无效的多字节字符。

- 不允许与服务器的区域设置不一致的数据库编码(Tom)

  在大多数平台上， `C` 区域是与任何数据库编码执行的唯一区域。 其它区域设置说明了一个特定的编码，并且如果数据库编码不同，则错误操作（典型的症状包括虚假文本 排序顺序和来自 `upper()` 或者 `lower()` 的错误结果） 现在服务器拒绝尝试创建有一个不兼容编码的数据库。

- 确保 `chr()` 不能创建无效编码值(Andrew)

在UTF8编码数据库中 `chr()` 参数被认为Unicode代码点。 在其它多字节编码中 `chr()` 的参数必须指定7位ASCII字符。 不再接受零。调整 `ascii()` 以达到匹配。

- 调整 `convert()` 操作用来确保编码有效性(Andrew)

  `convert()` 的两个参数形式已被删除。 这三个参数的形式现在需要 `bytea` 第一个参数，并返回一个 `bytea` 。 为了代替函数缺失，三个新的函数被添加：

  - `convert_from(bytea, name)` 返回 `text` — 从指定编码的第一个参数转化到数据库编码
  - `convert_to(text, name)` 返回 `bytea` —； 转换数据库编码的第一个参数到指定编码
  - `length(bytea, name)` 返回 `integer` — 给定指定编码字符中第一个参数长度

- 删除 `转换(参数使用conversion_name)` (Andrew)

  其操作不符合SQL标准。

- 客户端JOHAB编码(Tatsuo)

  JOHAB作为服务器端编码不安全。

E.77.3. 变化

下面你将发现在PostgreSQL 8.3和之前主要发布版本之间变化的详细说明。

E.77.3.1. 性能

- 在事务提交期间异步提交延迟写入WAL(Simon)

  此功能极大地提高了短期数据修改事务性能。 缺点是由于磁盘写入延迟， 如果数据被写入到磁盘之前该数据库或操作系统崩溃， 那么提交的数据将会丢失。此功能对可以接受一些数据丢失的应用程序是有益处的。 不像关闭 `fsync` ， 使用异步提交不会把数据库一致性放在危险的处境中; 最坏的情况是崩溃后最后几个据称已提交的事务可能不会被提交。 此功能通过关闭 `synchronous_commit` 被启用 （由每个会话或每个事务来完成，如果一些事务是关键的，而另一些则不是）。 `wal_writer_delay` 在事务实际到达磁盘之前可以被调整以控制最大延迟。

- 检查点写操作被分散在一个较长时间周期内用来平滑每个检查点中I/O剧增 (Itagaki Takahiro和Heikki Linnakangas)

  以前所有修改过的缓冲区在检查点期间尽可能快地被强制到磁盘中， 造成I/O剧增，降低服务器的性能。 这种新方法在检查点中向外扩散到磁盘写入， 降低最大I/O使用。（尽可能快的写入用户请求和关闭检查点）。

- 堆元组（HOT）为 `UPDATE` 和 `DELETE` 加速空间再利用(Pavan Deolasee, 还有许多其他的想法)

  `UPDATE` 和 `DELETE` 舍弃死元组， 因为执行失败的 `INSERT` 。之前仅仅 `VACUUM` 可以回收通过死元组采取的空间。 如果没有变化到索引列，那么在 `INSERT` 或者 `UPDATE` 的时候HOT死元组空间可以被自动回收。 这允许更加一致的性能。此外，HOT避免添加重复索引项。

- 实时后台写策略提高磁盘写入效率(Greg Smith, Itagaki Takahiro)

  这极大降低了后台写入的手动调整需要。

- 降低每个字段和每行存储开销(Greg Stark, Heikki Linnakangas)

  数据值小于128字节的可变长度数据类型将看到3至6个字节的存储减少。 例如，两个相邻的 `char(1)` 字段现在使用4个字节，而不是16。 行标头是比以前更短的4个字节。

- 为只读事务使用非持久事务ID减少开销和 `VACUUM` 需求(Florian Pflug)

  非持久性事务ID不会增加全局事务计数器。因此，它能减少 `pg_clog` 上负载 并且增加强制vacuum以防止事务ID重叠期间的时间。 采取的其它性能改进提高了并发性。

- 在只读命令后避免增加命令计数器(Tom)

  之前每个事务2³²(4十亿)命令的硬性限制。 现在唯一的命令实际上改变了数据库计数， 因此，虽然这一限制仍然存在，它应该很少令人讨厌。

- 创建专用WAL写进程从后台卸载操作(Simon)

- 为 `CLUSTER` 和 `COPY` 忽略不必要的WAL写进程(Simon)

  除非启用WAL归档，那么在命令结尾为 `CLUSTER` 和 `fsync()` 表系统避免WAL写操作。 如果在同一个事务中创建该表，那么与 `COPY` 是一样的。

- 大顺序扫描不再频繁删除已使用缓存页(Simon, Heikki, Tom)

- 并发大顺序扫描可以共享磁盘读取(Jeff Davis)

  通过在表的中间开始新的顺序扫描被完成 （其中另一个顺序扫描在进行中），并返回开始以完成。 这会影响没有指定 `ORDER BY` 的查询中返回行的顺序。 如果有必要的话， `synchronize_seqscans` 配置参数可以用来禁用它。

- `ORDER BY ... LIMIT` 无需排序(Greg Stark)

  这是通过顺序扫描表并且跟踪"top N"候选行执行， 而非执行全排序整个表。 当没有匹配索引并且 `LIMIT` 不大的时候，这是很有用的。

- 将信息上速度限制发送到后台统计收集器上(Tom)

这降低了短事务开销，但是可能有时在统计之前增加延迟。

- 完善许多空值情况下哈希连接性能(Tom)
- 加快非精确数据类型匹配操作符查找情况(Tom)

E.77.3.2. 服务器

- 缺省启用Autovacuum (Alvaro)

  进行了几处修改以消除启用自动清理的缺点， 从而证明缺省变化。一些其他自动清理参数缺省也被修改。

- 支持多个并发自动清理进程(Alvaro, Itagaki Takahiro)

  这允许同时运行多个清理。这可以阻止大表清理延迟小表清理。

- 当表定义变化或者统计被更新的时候，自动重新规划缓存查询(Tom)

  如果临时表在函数调用之间被删除并且重新创建， 之前引用临时表的PL/pgSQL函数可能会失败， 除非使用 `EXECUTE` 。 这个改进解决了该问题和许多相关的问题。

- 添加 `temp_tablespaces` 参数 以控制临时表和文件的表空间(Jaime Casanova,Albert Cervera, Bernd Helmle)

  该参数定义了要使用的表空间列表。 启用在多个表空间上分散I/O负载。 随机表空间在创建一个临时对象时被选择。 临时文件不再存储在每个数据库 `pgsql_tmp/` 目录中， 而在每个表空间目录中。

- 将临时表的TOAST表放在 `pg_toast_temp_``_nnn_` 命名的特殊模式中(Tom)

  这使得低级别的代码将这些表看作临时的， 这可以启动各种优化比如没有WAL日志记录更改 并且使用本地而不是共享缓存的访问。这也修复了一个错误， 其中后端意外保持打开的文件引用到临时TOAST表。

- 修复新连接请求恒流可能无限延迟postmaster完成关机或者死机重启的问题(Tom)
- 警惕非常低概率的数据丢失情况，防止 重新使用已删除表的relfilenode，直到下一个检查点之后(Heikki)
- 修复 `CREATE CONSTRAINT TRIGGER` 用来转换旧形式外键触发器定义 到规则外键约束(Tom)

  这将缓解来自7.3之前数据库外键约束移植，如果它们从来没使用 `contrib/adddepend` 被转移。

- 修复 `DEFAULT NULL` 重写继承的缺省值(Tom)

  `DEFAULT NULL` 之前被认为噪声短语，但是它应该（并且现在）覆盖非零缺省 否则从父表或者域中被继承。

- 添加新的编码EUC_JIS_2004和SHIFT_JIS_2004 (Tatsuo)

  这些新编码可以从UTF-8被转换。

- 从"数据库系统准备就绪"到"数据库系统准备接受连接" 改变服务器启动日志信息，并且调整其定时

  仅仅当postmaster准备好接受连接的时候出现该信息。

E.77.3.3. 监控

- 添加 `log_autovacuum_min_duration` 参数以支持autovacuum活动可配置记录(Simon, Alvaro)
- 添加 `log_lock_waits` 记录锁等待(Simon)
- 添加 `log_temp_files` 记录临时文件用法(Bill Moran)
- 添加 `log_checkpoints` 参数完善检查点记录(Greg Smith, Heikki)
- `log_line_prefix` 目前支持所有过程中 `%s` 和 `%c` 逃逸(Andrew)

  之前这些逃逸只用于用户会话中，而不是后台数据库过程。

- 添加 `log_restartpoints` 控制即时恢复重启点记录(Simon)
- 最后事务结束时间被记录到恢复结束和每个记录重启点(Simon)
- Autovacuum报告 `pg_stat_activity` 中活动启动时间(Tom)
- 允许逗号分隔值（CSV）形式的服务器日志输出(Arul Shaji, Greg Smith, Andrew Dunstan)

  CSV格式日志文件可以很容易为后续分析被加载到数据库表中。

- 使用PostgreSQL提供的时区支持服务器日志中显示的格式时间戳(Tom)

  这避免了提供错误编码中本地化时区名称Windows特定问题。 有一个新的 `log_timezone` 参数控制用于日志消息中的时区， 独立于客户端可见 `timezone` 参数。

- 新系统视图 `pg_stat_bgwriter` 显示关于后台写活动统计(Magnus)
- 为数据库端元组统计添加新列到 `pg_stat_database` (Magnus)
- 添加 `xact_start` (事务启动时间)列到 `pg_stat_activity` (Neil)

  这使得很容易识别长期存在的事务。

- 添加 `n_live_tuples` 和 `n_dead_tuples` 列到 `pg_stat_all_tables` 和相关视图中(Glen Parker)

- 合并 `stats_block_level` 和 `stats_row_level` 参数到单一参数 `track_counts`， 这控制着发送到统计收集器过程的所有信息(Tom)
- 重命名 `stats_command_string` 参数为 `track_activities` (Tom)
- 修复活的和死的元组统计计数以意识到已提交的和终止的事务有不同效果(Tom)

E.77.3.4. 认证

- 为Windows上认证支持安全服务提供者接口(SSPI) (Magnus)
- 支持GSSAPI 认证(Henry Hotz, Magnus)

  这是首先的本地Kerberos认证，因为GSSAPI是行业标准。
- 支持全球SSL配置文件(Victor Wagner)
- 添加 `ssl_ciphers` 参数控制可接受SSL加密(Victor Wagner)
- 添加Kerberos范围参数， `krb_realm` (Magnus)

E.77.3.5. 预写日志(WAL)和连续归档

- 改变来自time_t到TimestampTz表示形式的事务WAL记录中的时间戳记录(Tom)

  在WAL中提供了列居第二位解决方法。这对于点即时恢复有益处。
- 减少热备份服务器需要的WAL磁盘空间(Simon)

  这种变化允许热备用服务器传递最早仍然需要WAL文件名字到恢复脚本， 允许不再需要的WAL文件自动清除。 使用 `recovery.conf` 中 `restore_command` 参数 `%r` 的执行。
- 新的布尔配置参数， `archive_mode` ，控制归档(Simon)

  之前设置 `archive_command` 为空字符串关闭归档。现在 `archive_mode` 使归档开启和关闭，独立于 `archive_command` 。 这用于暂时停止归档。

E.77.3.6. 查询

- 全文搜索被集合到核心数据库系统中(Teodor, Oleg)

  文本搜索已被改进，移动到核心代码中，并且缺省安装。 `contrib/tsearch2` 包含兼容接口。
- 添加控制是否 `NULL` 首先排序还是最后排序(Teodor, Tom)

  语法是 `ORDER BY ... NULLS FIRST/LAST` 。

- 允许每列升序/降序（ `ASC` / `DESC` ）索引排序选项(Teodor, Tom)

  之前使用带有混合 `ASC` / `DESC` 说明符 `ORDER BY` 的查询 不能完全使用索引。如果使用匹配的 `ASC` / `DESC` 说明符创建索引， 那么现在在这种情况中可以完全使用索引。可以控制索引中 `NULL` 排序顺序。

- 允许 `col IS NULL` 使用索引(Teodor)
- 可更新游标(Arul Shaji, Tom)

  这消除了参考主键到游标返回的 `UPDATE` 或者 `DELETE` 行的需要。 语法为 `UPDATE/DELETE WHERE CURRENT OF` 。

- 允许游标中 `FOR UPDATE` (Arul Shaji, Tom)
- 创建通用机制为每个数据类型支持来自标准字符串类型(`TEXT` , `VARCHAR` , `CHAR`)转换，通过调用数据类型的I/O函数(Tom)

  之前，这样的转换只适用于此目的具有专门函数的类型。 这些新的类型转换只在字符串方向被分配，显式仅在其他方向， 因此应该创建不令人惊讶的操作。

- 当所有输入是域类型时，允许 `UNION` 和相关结构返回域类型(Tom)

  之前，输出可能被认为是域的基本类型。

- 当使用两种不同数据类型的时候，允许有限的散列(Tom)

  这允许哈希连接，哈希索引，散列子规划，以及 在涉及跨数据类型比较的情况下使用的散列聚合， 如果该数据类型具有兼容的散列函数。 目前，跨数据类型散列支持存在于 `smallint` / `integer` / `bigint` , 以及 `float4` / `float8` 中。

- 当 `WHERE` 子句中变量相等时，改善优化逻辑检测(Tom)

  这允许降序排序顺序合并连接，并且改进识别冗余排序列。

- 在大多数表通过约束被排除的情况下规划大继承树时，提高性能(Tom)

E.77.3.7. 对象操作

- 复合类型数组(David Fetter, Andrew, Tom)

  除了显式声明的复合类型的数组外， 目前支持常规表和视图行类型数组，除了系统目录行类型，序列和TOAST表。

- 每个函数基础上设置服务器配置参数(Tom)

  比如，如果在运行时存在不同的 `search_path` ，函数可以设置自身的 `search_path` 避免意外操作。安全定义函数可以设置 `search_path` 以避免安全漏洞。

- `CREATE/ALTER FUNCTION` 现在可以支持 `COST` 和 `ROWS` 选项(Tom)

  `COST` 允许函数调用成本的说明。 `ROWS` 允许平均数或者设置返回函数返回的行的说明。在选择最佳规划中通过优化器使用这些值。

- 实现 `CREATE TABLE LIKE ... INCLUDING INDEXES` (Trevor Hardcastle, Nikhil Sontakke, Neil)

- 允许 `CREATE INDEX CONCURRENTLY` 忽略其它数据库中事务(Simon)

- 添加 `ALTER VIEW ... RENAME TO` 和 `ALTER SEQUENCE ... RENAME TO` (David Fetter, Neil)

  之前这可能通过 `ALTER TABLE ... RENAME TO` 执行。

- 使得 `CREATE/DROP/RENAME DATABASE` 短暂等待冲突后端在失败之前退出(Tom)

  这增加了这些命令成功的可能性。

- 允许触发器和规则出于复制目的在使用配置参数组中无效(Jan)

  这允许复制系统禁用触发器并且重写规则作为无需直接修改系统目录组。 该操作通过 `ALTER TABLE` 和新参数 `session_replication_role` 被 控制。

- 用户定义类型可以有类型修饰符(Teodor, Tom)

  这允许用户定义类型采取修饰符，像 `ssnum(7)` 。 之前只有内置数据类型可以有修饰符。

E.77.3.8. 实用命令

- 非超级用户数据库所有者可以添加可信任程序语言到缺省数据库中(Jeremy Drake)

  虽然这是相当安全的，一些管理员可能希望撤销该权限。 通过 `pg_pltemplate` . `tmpldbacreate` 控制。

- 允许会话的当前参数设置作为将来会话缺省使用(Tom)

  在 `CREATE/ALTER FUNCTION` , `ALTER DATABASE` ， 或者 `ALTER ROLE` 中使用 `SET ... FROM CURRENT` 执行。

- 实现新的命令 `DISCARD ALL` , `DISCARD PLANS` ， `DISCARD TEMPORARY` ， `CLOSE ALL` 和 `DEALLOCATE ALL` (Marko Kreen, Neil)

  这些命令简化重置数据库会话到它的初始状态，并且特别有利于连接池软件。

- 使得 `CLUSTER` MVCC-安全(Heikki Linnakangas)

  之前， `CLUSTER` 可能丢弃所有已提交为死的元组， 即使仍然有事务可以在MVCC可见规则下看到它们。

- 添加 `CLUSTER` 语法: `CLUSTER` `_table_` USING `_index_` (Holger Schurig)

  仍然支持旧的 `CLUSTER` 语法，但是新形式更加合理。

- 修复 `EXPLAIN` 以致于它可以更加准确显示复杂规划(Tom)

  参考子规划输出总是正确被显示，而不是为复杂情况使用 `?column``_N_` ?。

- 当删除用户的时候，限制报导信息量(Alvaro)

  之前，删除（或尝试删除）拥有很多对象的用户 可能导致列出所有这些对象的大的 `NOTICE` 或者 `ERROR` 信息； 这造成一些客户端应用程序问题。信息长度是有限的， 但一个完整列表仍被发送到服务器日志。

E.77.3.9. 数据类型

- 支持SQL/XML标准，包含新的操作以及 `XML` 数据类型(Nikolay Samokhvalov, Pavel Stehule, Peter)

- 枚举数据类型(`ENUM`) (Tom Dunstan)

  这个特性提供了方便支持小的，不变设置允许值的字段。 创建 `ENUM` 类型的例子是 `CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')` 。

- 通用唯一标识符(`UUID`)数据类型(Gevik Babakhani, Neil)

  这紧密匹配RFC 4122。

- 加宽 `MONEY` 数据类型到64位(D'Arcy Cain)

  这大大增加了支持 `MONEY` 的范围

- 修复 `float4` / `float8` 以不断地处理 `Infinity` 和 `NAN` (非数字)(Bruce)

  之前代码关于从溢出条件中识别 `Infinity` 是不一致的。

- 在 `boolean` 值的输入中允许前导和尾随空格(Neil)

- 阻止 `COPY` 使用数字和大小写字母作为分隔符(Tom)

E.77.3.10. 函数

- 添加新规则表达式函数 `regexp_matches()` , `regexp_split_to_array()` 和 `regexp_split_to_table()` (Jeremy Drake, Neil)

  这些函数提供了正则表达式子表达式的提取，并且允许使用POSIX正则表达式分裂字符串。

- 为大对象截断添加 `lo_truncate()` (Kris Jurka)

- 为 `float8` 数据类型实现 `width_bucket()` (Neil)
- 添加 `pg_stat_clear_snapshot()` 用来丢弃当前事务中收集的统计快照(Tom)

  事务中统计的第一个请求需要事务期间不发生变化的统计快照。 此函数允许快照被丢弃，并且在接下来的统计查询中加载一个新的快照。 这对PL/pgSQL函数特别有用， 这些函数被限制在一个单一事务中。

- 添加 `isodow` 选项 `EXTRACT()` 和 `date_part()` (Bruce)

  这返回一周中一天，星期日为七。（ `dow` 返回星期日为零）

- 为 `to_char()`， `to_date()` 和 `to_timestamp()` 添加 `ID` (一周中ISO天)和 `IDDD` (一年中ISO天)格式代码(Brendan Jurd)
- 采用 `to_timestamp()` 和 `to_date()` 为潜在的可变宽度字段假设 `TM` (修剪)选项(Bruce)

  这匹配Oracle的操作。

- 修复 `to_date()` / `to_timestamp()``D` (一周中非ISO天)字段中偏移一个单位转换错误(Bruce)
- `setseed()` 返回空，而不是无效整数值(Neil)
- 为 `NUMERIC` 添加哈希函数(Neil)

  这允许哈希索引和基于哈希规划用于 `NUMERIC` 列。

- 提高 `LIKE` / `ILIKE` 的效率，尤其是 多字节字符集设置像UTF-8 (Andrew, Itagaki Takahiro)
- 使用 `currtid()` 函数需要目标表上 `SELECT` 权限(Tom)
- 添加几个 `txid_*()` 函数以查询活跃事务ID(Jan)

  这对于各种复制方法是有用的。

E.77.3.11. PL/pgSQL服务器端语言

- 添加可滚动游标支持，包含 `FETCH` 中方向控制(Pavel Stehule)
- 允许 `IN` 作为PL/pgSQL的 `FETCH` 语句中 `FROM` 的选择，与后端的 `FETCH` 命令一致(Pavel Stehule)
- 添加 `MOVE` 到PL/pgSQL (Magnus, Pavel Stehule,Neil)
- 实现 `RETURN QUERY` (Pavel Stehule, Neil)

这添加了想要返回查询结果的PL/pgSQL设置返回函数的便捷语法。 `RETURN QUERY` 比环绕 `RETURN NEXT` 更容易而且更有效。

- 允许函数参数名字符合函数的名字(Tom)

  比如 `myfunc.myvar` 。 在变量名可能匹配列名的查询中声明变量非常有用。

- 使得块标签变量正常执行(Tom)

  之前，外部级别块标签可能意外干扰内部级别记录或者行引用的识别。

- 严格控制 `FOR` 循环 `STEP` 值的需求(Tom)

  防止非负 `STEP` 值，并且处理循环溢出。

- 当报告语法错误位置的时候，提高精度(Tom)

E.77.3.12. 其它服务器端语言

- 允许类型名参数到PL/Perl `spi_prepare()` 成为数据类型别名， 除了 `pg_type` 中发现的名字(Andrew)
- 允许类型名参数到PL/Python `plpy.prepare()` 成为数据类型别名， 除了在 `pg_type` 中发现的名字(Andrew)
- 允许类型名参数到PL/Tcl `spi_prepare` 成为数据类型别名， 除了在 `pg_type` 中发现的名字(Andrew)
- 启动PL/PythonU在Python 2.5上编译(Marko Kreen)
- 支持在兼容的Python版本中（Python 2.3和之后版本）真正的PL/Python布尔类型 (Marko Kreen)
- 修复后台中线程启用 `libtcl` 产生大量线程的PL/Tcl问题 (Steve Marshall, Paul Bayer,Doug Knight)

  这导致了种种不愉快。

E.77.3.13. psql

- 分别列出 `\d` 输出中禁用触发器(Brendan Jurd)
- 在 `\d` 模式中，总是逐字匹配 `$` (Tom)
- 显示 `\da` 输出中聚合返回类型(Greg Sabino Mullane)
- 添加函数的波动状态到 `\df+` 的输出中(Neil)
- 添加 `\prompt` 性能(Chad Wagner)

- 添加 `\pset`，`\t` 和 `\x` 用来声明 `on` 或者 `off`， 而不仅仅是切换(Chad Wagner)
- 添加 `\sleep` 能力(Jan)
- 为 `\copy` 启用 `\timing` 输出(Andrew)
- 提高Windows上 `\timing` 方法(Itagaki Takahiro)
- 在每个反斜杠命令后冲洗 `\o` 输出(Tom)
- 当读取 `-f` 输入文件的时候，正确地检测和报告错误(Peter)
- 删除 `-u` 选项（该选项已经很长时间不使用）(Tom)

E.77.3.14. pg_dump

- 添加 `--tablespaces-only` 和 `--roles-only` 选项到pg_dumpall (Dave Page)
- 添加输出文件选项到pg_dumpall (Dave Page)

  这在Windows上有用，子pg_dump进程输出重定向不执行。
- 允许pg_dumpall接受初始连接数据库名而不是缺省 `template1` (Dave Page)
- 在 `-n` 和 `-t` 切换中，总是逐字匹配 `$` (Tom)
- 当数据库中有成千上万对象时提高性能(Tom)
- 删除 `-u` 选项（该选项已经很长时间不使用）(Tom)

E.77.3.15. 其它客户端应用程序

- 在initdb中，允许指定 `pg_xlog` 目录位置(Euler Taveira de Oliveira)
- pg_regress中支持的操作系统上启用服务器核心转储生成(Andrew)
- 添加 `-t` (超时)参数到pg_ctl(Bruce)

  当等待服务器启动或者关闭的时候，这将控制pg_ctl等待的多长时间。 之前超时被硬连线为60秒。
- 添加pg_ctl选项来控制服务器核心转储生成(Andrew)
- 允许控制C取消clusterdb，reindexdb和vacuumdb (Itagaki Takahiro, Magnus)
- 为createdb，createuser，dropdb和 dropuser抑制命令标签输出(Peter)

  忽略 `--quiet` 选项，并且在8.4中删除。 当在数据库上操作定位到标准输出上而非标准错误上时，促进信息，因为它们并不是真正错误。

E.77.3.16. libpq

- 如果它包含一个等号，那么解析 `PQsetdbLogin()` 的 `dbName` 参数作为 `conninfo` 字符串(Andrew)

  这允许客户端程序中 `conninfo` 字符串的使用仍然使用 `PQsetdbLogin()` 。

- 支持全局SSL配置文件(Victor Wagner)
- 添加环境变量 `PGSSLKEY` 支持SSL硬件密钥(Victor Wagner)
- 为大对象截断添加 `lo_truncate()` (Kris Jurka)
- 如果服务器需要密码但是没有提供，那么添加 `PQconnectionNeedsPassword()` 返回真(Joe Conway, Tom)

  如果失败连接尝试后返回真，那么客户端应用程序应该提示用户输入密码。 以往应用程序不得不检查一个特定的错误消息字符串 以决定是否需要密码；这种方法现在已经弃用。

- 如果提供的密码已经被使用，那么添加 `PQconnectionUsedPassword()` 返回真(Joe Conway, Tom)

  这在一些安全的地方是有用的，知道用户提供的密码是否有效非常重要。

E.77.3.17. ecpg

- 使用V3 前端/后端协议(Michael)

  这添加了对服务器端预备语句的支持。

- 使用本地支持，而不是Windows上pthreads(Magnus)
- 提高ecpglib的线程安全(Itagaki Takahiro)
- 使得ecpg库输出必要的API符号(Michael)

E.77.3.18. Windows端口

- 允许使用Microsoft Visual C++编译整个PostgreSQL发布(Magnus 和其他人)

  这允许基于Windows的开发人员可以使用熟悉的开发和调试工具。 用Visual C++制作的Windows可执行文件比用其他工具集制作的可能有更好的稳定性和性能。 已经删除仅仅客户端的Visual C++编译脚本。

- 当有许多子进程的时候，大大减少postmaster的内存使用(Magnus)
- 允许通过管理用户启动回归测试(Magnus)

- 添加本地共享内存实现(Magnus)

E.77.3.19. 服务器编程接口(SPI)

- 在SPI中添加游标相关功能(Pavel Stehule)

  允许访问游标相关规划选项，并且添加 `FETCH` / `MOVE` 程序。

- 允许通过 `SPI_execute` 执行游标命令(Tom)

  宏 `SPI_ERROR_CURSOR` 仍然存在但永远不会被返回。

- SPI规划指针被声明为 `SPIPlanPtr` 而不是 `void *` (Tom)

  这不会破坏应用程序代码，但是推荐切换来帮助捕获简单程序错误。

E.77.3.20. 编译选项

- 添加configure选项 `--enable-profiling` 用来启动代码分析(使用gcc执行)(Korry Douglas 和Nikhil Sontakke)
- 添加configure选项 `--with-system-tzdata` 用来 使用操作系统的时区数据库(Peter)
- 修复PGXS，以致于扩展会不利于PostgreSQL的安装，其中 pg_config程序没有首先出现在 `PATH` 中(Tom)
- 当构建SGML文档的时候，支持 `gmake draft` (Bruce)

  除非使用 `draft` ，如果确保索引是最新的，那么文档建立将被重复。

E.77.3.21. 源代码

- 重命名宏 `DLLIMPORT` 到 `PGDLLIMPORT` 以避免与包含（像Tcl）定义 `DLLIMPORT` 的第三方冲突 (Magnus)
- 创建"操作符类"用来改善涉及交叉数据类型比较的查询规划(Tom)
- 更新GIN `extractQuery()` API允许发送信号没什么可以满足查询(Teodor)
- 删除从 `postgres_ext.h` 到 `pg_config_manual.h` 的 `NAMEDATALEN` 定义(Peter)
- 在所有平台上提供 `strlcpy()` 和 `strlcat()` ， 并且替换 `strncpy()` ， `strncat()` 的易出错的用法，等(Peter)
- 创建钩子以便使外部插件监听（或者甚至替换）规划器并且为假设情况创建规划(Gurjeet Singh, Tom)
- 创建函数变量 `join_search_hook` 让插件覆盖规划器的连接搜索顺序部分(Julius Stroffek)

- 添加 `tas()` 支持Renesas的M32R处理器(Kazuhiro Inaoka)
- `quote_identifier()` 和pg_dump 不再引用依据语法未保留的关键字(Tom)
- 改变 `NUMERIC` 数据类型的磁盘上的表示，以致于 `sign_dscale` 词在权重之前(Tom)
- 使用SYSV信号而不是SYSV >= 6.0,即OS X 10.2及以上的POSIX(Chris Marcellino)
- 添加acronym和 NFS文档部分(Bruce)
- "Postgres"作为"PostgreSQL"的公认别名(Peter)
- 当服务器关闭的时候，添加关于避免数据库服务器欺骗的文档(Bruce)

E.77.3.22. Contrib

- 移动 `contrib` `README` 内容到主PostgreSQL文档 (Albert Cervera i Areny)
- 为低级别页面检查添加 `contrib/pageinspect` (Simon, Heikki)
- 为控制热备份操作添加 `contrib/pg_standby` 模块(Simon)
- 为使用OSSP UUID库生成 `UUID` 值添加 `contrib/uuid-ossp` 模块(Peter)

  使用configure `--with-ossp-uuid` 激活， 这利用新的 `UUID` 内置类型。
- 添加 `contrib/dict_int` ， `contrib/dict_xsyn` 和 `contrib/test_parser` 模块提供样本的附加文本搜索词典模板和解析器(Sergey Karpov)
- 允许contrib/pgbench设置填充因子(Pavan Deolasee)
- 添加时间戳到contrib/pgbench `-l` (Greg Smith)
- 添加使用次数统计到 `contrib/pgbuffercache` (Greg Smith)
- 为 `contrib/hstore` 添加GIN支持(Teodor)
- 为 `contrib/pg_trgm` 添加GIN支持(Guillaume Smet, Teodor)
- 在 `contrib/start-scripts` 中更新OS/X启动脚本(Mark Cotner, David Fetter)
- 限制 `pgrowlocks()` 和 `dblink_get_pkey()` 到在目标表上拥有 `SELECT` 权限的用户中(Tom)
- 限制 `contrib/pgstattuple` 函数到超级用户(Tom)
- `contrib/xml2` 过时了，计划在8.4中删除(Peter)

  核心PostgreSQL中新的XML支持取代该模块。

E.78. 版本 8.2.23

> 发布日期: 2011-12-05

这个版本包含各种自8.2.22以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

这预计是8.2.X系列的最后一个PostgreSQL版本。 建议用户尽快更新到一个新的版本分支。

E.78.1. 迁移到版本 8.2.23

运行8.2.X的用户不需要转储/恢复。

不过，在 `information_schema.referential_constraints` 视图的定义中发现一个长期存在的错误。如果你依赖于该视图的正确结果， 你应该像下面解释的第一个更新日志项一样替换它的定义。

还有，如果你是从一个早于8.2.14的版本升级而来，那么请查看8.2.14的版本声明。

E.78.2. 修改列表

- 修复在 `information_schema.referential_constraints` 视图中的错误 (Tom Lane)

  这个视图在匹配外键约束到决定性的主键或唯一键约束上不够细心。 这会导致未能显示外键约束，或者多次显示外键约束， 或者抱怨它依赖于一个与它实际依赖的不同的约束。

  因为视图定义是通过initdb安装的，只是升级将不能修复该问题。 如果你需要在一个现有的安装中修复该问题，你可以（作为超级用户） 删除 `information_schema` 模式，然后通过来源 `_SHAREDIR_` /information*schema.sql* 重新创建它。 （如果不确定 `_SHAREDIR* `在哪里，运行` pg_config --sharedir`。）必须在每个要修复的数据库中重复该步骤。

- 修复 `CREATE TABLE dest AS SELECT * FROM src` 或 `INSERT INTO dest SELECT * FROM src` 期间TOAST相关的数据损坏 (Tom Lane)

  如果一个表通过 `ALTER TABLE ADD COLUMN` 修改了， 那么尝试逐字的拷贝它的数据到另外一个表可能在极端情况下产生损坏的结果。 这个问题只能在8.4及以后的版本中以这个精确的形式来验证， 不过我们也给早期的版本打了补丁，以防有其他的代码路径会触发相同的错误。

- 修复toast表访问陈旧的系统缓存记录时的竞态条件 (Tom Lane)

典型的症状是像"missing chunk number 0 for toast value NNNNN in pg_toast_2619" 这样的瞬态错误，这里引用的toast表将总是属于一个系统目录。

- 改善 `money` 类型的输入和输出中的本地支持 (Tom Lane)

  除了不支持所有标准的 `lc_monetary` 格式选项之外，输入和输出函数是不考虑的，意味着有的环境中转储的 `money` 值不能被重新读取。

- 让 `transform_null_equals` 不影响 `CASE foo WHEN NULL ...` 结构 (Heikki Linnakangas)

  `transform_null_equals` 只应该影响直接由用户编写的 `foo = NULL` 表达式，不等于 `CASE` 的这个格式内部产生的检查。

- 修改外键触发器创建，要求更好的支持自我引用的外键 (Tom Lane)

  对于一个引用自身的表的级联外键，一个行更新将作为一个事件触发 `ON UPDATE` 触发器和 `CHECK` 触发器。必须先执行 `ON UPDATE` 触发器， 否则 `CHECK` 将检查一个行的非最终状态，并且可能抛出一个不合适的错误。 然而，这些触发器的触发顺序是通过他们的名字决定的，通常以特定的顺序排序， 因为触发器有自动生成的名字，遵从约定"RI_ConstraintTrigger_NNNN"。 修改该约定将需要一个适当的修复，我们将在9.2中实现， 但是在现有的版本中修改它似乎是危险的。所以这个路径只是改变了触发器的创建顺序。 用户遇到这个类型的错误时应该删除并重建外键约束，以使它的触发器的顺序正确。

- 保护psql命令历史中的命令中的空行 (Robert Haas)

  例如，如果从一个字符串文本中删除了一个空行，前者的行为会导致问题。

- 使用xsubpp的首选版本建立PL/Perl， 不一定是操作系统的主要拷贝 (David Wheeler 和 Alex Hunsaker)

- 遵从 `pgstatindex()` 中的查询取消立即中断 (Robert Haas)

- 确保VPATH构建适当的安装所有的服务器头文件 (Peter Eisentraut)

- 缩短在详细错误消息中报告的文件名 (Peter Eisentraut)

  常规的构建总是只是报告包含错误消息调用的C文件的名字， 但是VPATH构建以前报告一个绝对路径名。

- 为Central America修复Windows时区名的解释 (Tom Lane)

  映射"Central America Standard Time"到 `CST6` ， 而不是 `CST6CDT` ，因为DST通常没有观察到Central America的每一个地方。

- 更新时区数据文件到tzdata版本2011n，因为DST规律在 Brazil、Cuba、Fiji、Palestine、Russia和Samoa发生了改变 ； 还为Alaska和British East Africa做了历史纠正。

E.79. 版本 8.2.22

> 发布日期: 2011-09-26

这个版本包含各种自8.2.21以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

PostgreSQL社区将在2011年12月停止对8.2.X版本系列发布更新。 建议用户尽快更新到一个新的版本分支。

E.79.1. 迁移到版本 8.2.22

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.79.2. 修改列表

- 修复GiST索引页分裂处理中的多个错误 (Heikki Linnakangas)

  发生的可能性很低，但是会导致索引损坏。

- 避免 `ANALYZE` 中可能的访问关闭内存的结尾 (Noah Misch)

  这修复了一个非常低可能的服务器崩溃情况。

- 修复relcache初始化文件失效中的竞态条件 (Tom Lane)

  新的后端进程中有一个窗口会读取陈旧的初始化文件， 但是错过会告诉它该数据是陈旧的的inval消息。 结果会是目录访问中奇怪的失败，典型的是在启动后 "could not read block 0 in file ..."。

- 修复GiST索引扫描结束时的内存泄露 (Tom Lane)

  执行许多独立GiST索引扫描的命令，比如在一个早已包含许多行的表上新建基于GiST的排除约束的验证， 由于这个泄露，可能会在瞬间需要大量的内存。

- 修复构建一个大型的、有损耗的位图时的性能问题 (Tom Lane)

- 修复数组和路径创建函数，确保填充字节是零 (Tom Lane)

  这避免了一些规划器认为语义上相等的常数不等，导致欠佳的最优化的情况。

- 绕开中断WAL重放的gcc 4.6.0 bug (Tom Lane)

这可能导致在服务器崩溃后丢失已提交的事务。

- 修复一个视图中 `VALUES` 的转储bug (Tom Lane)
- 不允许在一个序列上 `SELECT FOR UPDATE/SHARE` (Tom Lane)

  这个操作不想预期的那样工作，并且会导致失败。

- 计算一个哈希表的大小时，防止整数溢出 (Tom Lane)
- 修复"peer"认证的证书控制消息中正在使用的潜在的bug (Tom Lane)
- 修复 `pg_srand48` 种子的初始化中的打印错误 (Andres Freund)

  这导致未能使用提供的种子的所有位。这个函数没有在大多数平台上使用 （只有那些没有 `srandom` 的平台），并且在任何情况下， 比预期的随机少的多下种子似乎很小，有潜在的安全风险。

- 当 `LIMIT` 和 `OFFSET` 值的和超过2^63时， 避免整数溢出 (Heikki Linnakangas)
- 添加溢出检查到 `generate_series()` 的 `int4` 和 `int8` 版本 (Robert Haas)
- 修复 `to_char()` 中的尾随零的删除 (Marti Raudsepp)

  在 `FM` 并且小数点后没有数字位的格式中，小数点左侧的零可能会被错误的删除。

- 修复 `pg_size_pretty()` ，避免接近2^63的输入溢出 (Tom Lane)
- 修复从一个不同的文件 `COPY` 期间，psql 的脚本文件行号的计数 (Tom Lane)
- 为 `standard_conforming_strings` 修复pg_restore 的直接到数据库模式 (Tom Lane)

  当从一个 `standard_conforming_strings` 设置为 `on` 制作的归档文件中直接恢复到一个数据库服务器时，pg_restore 可能发出不正确的命令。

- 修复libpq的LDAP服务查找代码中的过去写缓冲区结束和内存泄露 (Albe Laurenz)
- 在libpq中，避免使用非闭塞的I/O和SSL连接时的失败 (Martin Pihlak, Tom Lane)
- 改善连接启动期间libpq对失败的处理 (Tom Lane)

  特别的，SSL连接启动期间响应 `fork()` 失败的服务报告现在更加理智。

- 让ecpglib用15位精度写 `double` 值 (Akira Kurosawa)
- 为blowfish带符号字符bug应用逆向修复 (CVE-2011-2483)

  `contrib/pg_crypto` 的blowfish加密代码在字符是有符号的平台上 （大多数是这样）会给出错误的结果，导致加密的口令比应该的要弱。

- 修复 `contrib/seg` 中的内存泄露 (Heikki Linnakangas)

- 修复`pgstatindex()`，为空索引给出一致的结果 (Tom Lane)
- 允许使用perl 5.14建立 (Alex Hunsaker)
- 为系统函数存在的检测更新配置脚本的方法 (Tom Lane)

  我们在8.3和8.2中使用的autoconf版本会被执行连接时优化的编译器愚弄。
- 修复建立和安装文件路径包含空格的各种问题 (Tom Lane)
- 更新时区数据文件到tzdata版本2011i， 因为DST规律在Canada、Egypt、Russia、Samoa和South Sudan发生了改变。

E.80. 版本 8.2.21

> 发布日期: 2011-04-18

这个版本包含各种自8.2.20以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.80.1. 迁移到版本 8.2.21

运行8.2.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.80.2. 修改列表

- 避免目录缓存初始化期间潜在的死锁 (Nikhil Sontakke)

  在某些情况下，缓存加载代码会在锁定索引的目录之前在系统索引上请求共享锁。 这可能死锁尝试请求排他锁的进程，更标准的顺序。

- 当有并发的更新到目标行时，修复 `BEFORE ROW UPDATE` 触发器处理中的悬挂指针问题 (Tom Lane)

  已经观察到这个bug导致在尝试执行 `UPDATE RETURNING ctid` 时， 间歇的"cannot extract system attribute from virtual tuple"失败。 有非常小的可能会有更加严重的错误，比如为更新的元组产生不正确的索引项。

- 当表有等待延迟触发器事件时，不允许 `DROP TABLE` (Tom Lane)

  以前 `DROP` 会通过，导致触发器最终触发时 "could not open relation with OID nnn"错误。

- 修复包含数组切片的PL/Python内存泄露 (Daniel Popowich)

- 修复pg_restore以处理TOC文件中的长行(超过 1KB) (Tom Lane)

- 针对由于过度热情的编译器优化被零除引起的崩溃投入更多的保障 (Aurelien Jarno)

- 支持在FreeBSD和OpenBSD中的MIPS上的使用dlopen() (Tom Lane)

  有一个硬件连接的假设，这些系统函数在这些系统的MIPS硬件上不可用。 使用一个编译时测试替代，因为最近的版本已经可以了。

- 修复HP-UX上的编译失败 (Heikki Linnakangas)

- 修复Crywin上pg_regress使用的路径分隔符 (Andrew Dunstan)

- 更新时区数据文件到tzdata版本2011f， 因为DST规律在Chile、Cuba、Falkland Islands、Morocco、Samoa和Turkey发生了改变； 还为South Australia、Alaska和Hawaii做了历史纠正。

E.81. 版本 8.2.20

> 发布日期: 2011-01-31

这个版本包含各种自8.2.19以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.81.1. 迁移到版本 8.2.20

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.81.2. 修改列表

- 避免 `EXPLAIN` 尝试显示一个 `CASE` 表达式的简单格式时的失败 (Tom Lane)

  如果 `CASE` 的测试表达式是一个常量，规划器将简化 `CASE` 为一个混淆表达式显示代码的格式， 导致"unexpected CASE WHEN clause"错误。

- 修复分配给数组切片的下标在现有范围之前 (Tom Lane)

  如果在新添加的下标和已经存在的下标之间存在间隔， 代码错误计算有多少项需要从老的数组的空位图中拷贝， 潜在的导致数据损坏或崩溃。

- 避免规划器中非常遥远的日期值意外的转换溢出 (Tom Lane)

  `date` 类型比 `timestamp` 类型可以表示的日期支持的范围更加广泛， 但是规划器假设它总是可以将日期转换为时间戳而不受惩罚。

- 修复启用 `standard_conforming_strings` 时， pg_restore为大对象(BLOBs)的文本输出 (Tom Lane)

  尽管直接恢复到一个数据库能正确的工作，但是如果pg_restore 要求SQL文本输出和 `standard_conforming_strings` 已经在源数据库中启用了的话， 字符串的逃逸是不正确的。

- 修复包含 `... & !(subexpression) | ...` 的 `tsquery` 值的错误解析 (Tom Lane)

  包含这种操作符组合的查询没有正确的执行，相同的错误存在于 `contrib/intarray` 的 `query_int` 类型和 `contrib/ltree` 的 `ltxtquery` 类型中。

- 为 `query_int` 类型修复 `contrib/intarray` 的输入函数中的缓冲区溢出 (Apple)

这个错误是一个安全风险，因为该函数的返回地址可能会被重写。 感谢Apple Inc的安全团队报告这个问题并且提供该修复。 (CVE-2010-4015)

- 修复`contrib/seg`的GiST picksplit算法中的bug (Alexander Korotkov)

  这会导致相当大的低效，尽管不是真的错误回复，在`seg`字段的GiST索引中。 如果你有这样的一个索引，考虑在安装这个更新之后`REINDEX`它。 （这和在以前的更新中的`contrib/cube`中修复的错误相同。）

E.82. 版本 8.2.19

> 发布日期: 2010-12-16

这个版本包含各种自8.2.18以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.82.1. 迁移到版本 8.2.19

运行8.2.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.82.2. 修改列表

- 强制Linux上 `wal_sync_method` 的缺省为 `fdatasync` (Tom Lane, Marti Raudsepp)

  在Linux上的该缺省实际上是 `fdatasync` 已经很多年了， 但是最近的内核更改导致PostgreSQL选择了 `open_datasync` 。 这个选择没有导致任何性能改善，并且在某些文件系统上引起彻底的失败， 尤其是带有 `data=journal` 挂载选项的 `ext4` 。

- 为GIN索引修复WAL重放逻辑中的各种错误 (Tom Lane)

  这可能在复制期间导致"bad buffer id: 0"失败或索引内容的损坏。

- 当开始检查点WAL记录和它的重做点不在相同的WAL段时，修复从基础备份的恢复 (Jeff Davis)

- 添加对检测 `IA64` 上寄存器堆栈溢出的支持 (Tom Lane)

  `IA64` 体系结构有两个硬件堆栈。堆栈溢出失败的全面预防需要两个堆栈都检查。

- 为 `copyObject()` 中的堆栈溢出添加检查 (Tom Lane)

  由于堆栈溢出给出一个足够复杂的查询，某些代码路径可能会崩溃。

- 修复临时GiST索引中页分裂的检测 (Heikki Linnakangas)

  在一个临时索引中有"并发的"页分裂是可能的，比如说， 执行一个插入时有一个打开的游标扫描索引。GiST未能检测这个情况， 并且因此在该游标的执行继续时会交付错误的结果。

- 当 `ANALYZE` 复杂索引表达式时，避免内存泄露 (Tom Lane)

- 确保使用整行变量的索引仍然依赖于它的表 (Tom Lane)

像 `create index i on t (foo(t.*))` 这样声明的索引， 在它的表被删除时，将不会自动被删除。

- 不要用多个 `OUT` 参数"inline"一个SQL函数 (Tom Lane)

  这避免了由于丢失预期的结果行类型的信息而引起的可能的崩溃。

- 如果 `ORDER BY` 、 `LIMIT` 、 `FOR UPDATE` 或 `WITH` 附属于 `INSERT ... VALUES` 的 `VALUES` 部分则正确的行为 (Tom Lane)

- 修复 `COALESCE()` 表达式的常量折叠 (Tom Lane)

  规划器有时会尝试计算实际永远不可能达到的子表达式，可能导致意外的错误。

- 为 `InhRelation` 节点添加打印功能 (Tom Lane)

  这避免了启用 `debug_print_parse` 并且执行了某些类型的查询时的失败。

- 修复点到水平线段的距离的不正确的计算 (Tom Lane)

  这个错误影响几个不同的几何距离测量操作。

- 修复PL/pgSQL对"简单"表达式的处理， 在递归或错误恢复的情况下不会失败 (Tom Lane)

- 修复PL/Python对设置返回函数的处理 (Jan Urbanski)

  尝试在迭代器中调用SPI函数生成一组结果将会失败。

- 修复 `contrib/cube` 的GiST picksplit算法中的错误 (Alexander Korotkov)

  这会导致相当大的低效，尽管不是实际上不正确的答案，在一个 `cube` 字段的GiST索引中。 如果你有这样的一个索引，考虑在安装这个更新之后 `REINDEX` 它。

- 不要在 `contrib/dblink` 中发出"标识符将被截断"的通知， 除非创建新的连接 (Itagaki Takahiro)

- 修复 `contrib/pgcrypto` 中丢失的公共键上潜在的内核转储 (Marti Raudsepp)

- 修复 `contrib/xml2` 的XPath查询函数中的内存泄露 (Tom Lane)

- 更新时区数据文件到tzdata版本2010o， 因为DST规律在Fiji和Samoa发生了改变；还为Hong kong做了历史纠正。

E.83. 版本 8.2.18

> 发布日期: 2010-10-04

这个版本包含各种自8.2.17以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.83.1. 迁移到版本 8.2.18

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.83.2. 修改列表

- 为每个在PL/Perl和PL/Tcl中调用的SQL userid使用一个单独的解释器 (Tom Lane)

  这个修改阻止了通过破坏稍后由另一个SQL用户身份在同一个会话中执行的Perl或Tcl 代码引起的安全问题（例如，在一个 `SECURITY DEFINER` 函数中）。 大多数脚本语言提供多种可能实现的方式，比如重新定义被目标函数调用的标准函数或操作符。 没有这个修改，任何拥有Perl或Tcl语言使用权限的SQL用户基本上都可以用目标函数所有者的SQL权限做任何事情。

  这个修改的代价是Perl和Tcl函数之间有意的沟通变得更加困难。 为了提供一个安全出口，PL/PerlU和PL/TclU函数继续每个会话只使用一个解释器。 这不认为是一个安全问题，因为所有这样的函数都早已在数据库超级用户的信任级别执行。

  有可能第三方过程语言提供的受信任的执行有相似的安全问题。 我们建议为了安全鉴定的目的，联系任何你依赖的PL的作者。

  感谢Tim Bunce指出这个问题 (CVE-2010-3433)。

- 通过不允许 `pg_get_expr()` 被一个不是系统目录字段的参数调用， 阻止它中可能的崩溃 (Heikki Linnakangas, Tom Lane)

- 修复Windows共享内存分配代码 (Tsutomu Yamada, Magnus Hagander)

  这个错误导致经常报道"不能重新连接上共享内存"错误消息。 这是一个后向修复，前段时间已经应用到了新的分支。

- 在Windows上将退出代码128(`ERROR_WAIT_NO_CHILDREN`)看做非严重错误 (Magnus Hagander)

在高负载下，Windows进程有时会带有这个错误代码在启动时失败。 以前主进程将它看做一个恐慌条件，并重启整个数据库，但是这样看起来像是过度反应。

- 修复可能的 `UNION ALL` 成员关系的重复扫描 (Tom Lane)
- 修复"不能处理未计划的子查询"错误 (Tom Lane)

  这在子查询包含一个扩展成包含另一个子查询的表达式的连接别名引用时发生。

- 降低某些偶然报告的btree失败情况中的PANIC为ERROR， 并在结果错误消息中提供额外的详细信息 (Tom Lane)

  这会提高系统统损坏索引的鲁棒性。

- 阻止show_session_authorization()在自动清理进程中崩溃 (Tom Lane)
- 防御函数返回的记录集中不是所有返回的行都是相同的行类型 (Tom Lane)
- 修复哈希一个通过引用传递的函数结果时可能的失败 (Tao Ma, Tom Lane)
- 在写入锁文件时，小心同步锁文件的内容（ `postmaster.pid` 和套接字锁文件） (Tom Lane)

  如果机器在主进程启动之后不久就崩溃了，那么这个疏忽会导致损坏锁文件的内容。 反过来阻止随后启动主进程的尝试成功，直到手动删除了锁文件。

- 避免指定XID到深度嵌套的子事务时递归 (Andres Freund, Robert Haas)

  如果堆栈空间有限，那么原先的代码会导致崩溃。

- 修复 `log_line_prefix` 的 `%i` 逃逸， 它可能在后端启动时就产生垃圾 (Tom Lane)
- 修复启用归档时， `ALTER TABLE ... SET TABLESPACE` 中可能的数据损坏 (Jeff Davis)
- 允许 `CREATE DATABASE` 和 `ALTER DATABASE ... SET TABLESPACE` 被查询取消中断 (Guillaume Lelarge)
- 在PL/Python中，防御来自 `PyCObject_AsVoidPtr` 和 `PyCObject_FromVoidPtr` 的空指针结果 (Peter Eisentraut)
- 改善 `contrib/dblink` 对包含删除字段的表的处理 (Tom Lane)
- 修复 `contrib/dblink` 中"重复的连接名" 错误之后的连接泄露 (Itagaki Takahiro)
- 修复 `contrib/dblink` ，正确的处理长于62字节的连接名 (Itagaki Takahiro)
- 添加 `hstore(text, text)` 函数到 `contrib/hstore` (Robert Haas)

推荐这个函数作为现在废弃的 `=>` 操作符的替代品。 它是后向修复的，所以不会过时的代码可以使用老的服务器版本。 请注意，该修复将只在特定的数据库中安装或重新安装了 `contrib/hstore` 之后生效。用户可能更喜欢手动执行 `CREATE FUNCTION` 命令。

- 更新建立的基础结构和文档，反应源代码资源库从CVS搬到了Git (Magnus Hagander and others)

- 更新时区数据文件到tzdata版本20101， 因为DST规律在Egypt和Palestine发生了改变；还为Finland做了历史纠正。

  这个修改还为两个Micronesian时区添加了新的名字： Pacific/Chuuk现在优先于Pacific/Truk（首选的缩写是CHUT不是TRUT）， 和Pacific/Pohnpei优先于Pacific/Ponape。

- 让Windows的"N. Central Asia Standard Time"时区映射到Asia/Novosibirsk， 而不是Asia/Almaty (Magnus Hagander)

  微软在KB976098的时区更新中修改了这个时区的DST行为。 Asia/Novosibirsk是它的新行为的最好的匹配。

E.84. 版本 8.2.17

> 发布日期: 2010-05-17

这个版本包含各种自8.2.16以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.84.1. 迁移到版本 8.2.17

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.84.2. 修改列表

- 使用一个开放标记而不是使用 `Safe.pm` 应用到整个解释器， 在 `plperl` 中实施限制 (Tim Bunce, Andrew Dunstan)

  近期的发展已经向我们证实： `Safe.pm` 不够安全，不能依赖它标记 `plperl` 是可信任的。 这个修改一起删除了对 `Safe.pm` 的使用，为了支持总是使用一个带有开放代码标记的单独的解释器。 该修改令人愉快的一面包括：在 `plperl` 中以自然的方式使用Perl的 `strict` 编程现在是可能的了，并且Perl的 `$a` 和 `$b` 变量在短例程中像预期的那样工作了，并且函数的编译显著的更快了。 (CVE-2010-1169)

- 阻止PL/Tcl执行来自 `pltcl_modules` 的不可靠的代码 (Tom)

  PL/Tcl从数据库表自动加载Tcl代码的特性会被特洛伊木马攻击利用， 因为没有限制谁可以创建或插入到那个表。这个修改禁用了该特性， 除非 `pltcl_modules` 属于超级用户。 （不过，该表上的权限是没有检查的， 所以真实需要较少安全模块表的安装仍然可以赋予适当的权限给受信任的非超级用户。） 另外，阻止加载代码到无限制的"普通"Tcl解释器， 除非我们真的想要执行 `pltclu` 函数。 (CVE-2010-1170)

- 如果在重建relcache项期间接收到一个缓存重置消息，修复可能的崩溃 (Heikki)

  这个错误是在8.2.16中修复相关的失败的时候引入的。

- 不允许非特权的用户重置超级用户仅有的参数设置 (Alvaro)

  以前，如果一个非特权的用户为他自己运行 `ALTER USER ... RESET ALL`， 或为他拥有的一个数据库运行 `ALTER DATABASE ... RESET ALL`， 将会为该用户或数据库删除所有特殊参数设置，即使是只应该超级用户修改的设置。 现在， `ALTER` 将只删除该用户有权限修改的参数。

- 如果关机发生在 `CONTEXT` 添加到日志项时，避免后端关闭期间可能的崩溃 (Tom)

  在某些情况下，内容打印函数将会失败，因为当前事务在它想要打印日志消息时早已回滚了。

- 为现代的Perl版本更新pl/perl的 `ppport.h` (Andrew)
- 修复pl/python中的各种内存泄露 (Andreas Freund, Tom)
- 当扩展一个引用自身的变量时，阻止psql中的无限递归 (Tom)
- 修复psql的 `\copy` ，在 `\copy (select ...)` 的句点周围不添加空格 (Tom)

  在数值文字的小数点周围添加空格会导致语法错误。

- 确保 `contrib/pgstattuple` 函数迅速响应取消中断 (Tatsuhito Kasahara)
- 让服务器启动正确的处理 `shmget()` 为一个现有的共享内存段返回 `EINVAL` 的情况 (Tom)

  这个行为已经在BSD驱动的内核上（包括OS X）观察到了。 它导致一个完全误导的启动失败抱怨：共享内存请求尺寸过大。

- 避免Windows上系统日志处理中可能的崩溃 (Heikki)
- 更强劲的处理Windows注册中不完整的时区信息 (Magnus)
- 更新一组已知的Windows时区名 (Magnus)
- 更新时区数据文件到tzdata版本2010j， 因为DST规律在Argentina、Australian Antarctic、Bangladesh、Mexico、Morocco、 Pakistan、Palestine、Russia、Syria、Tunisia发生了改变； 还为Taiwan做了历史纠正。

  另外，添加 `PKST` (Pakistan Summer Time)到缺省的时区缩写集合。

E.85. 版本 8.2.16

> 发布日期: 2010-03-15

这个版本包含各种自8.2.15以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.85.1. 迁移到版本 8.2.16

运行8.2.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.85.2. 修改列表

- 添加新的配置参数 `ssl_renegotiation_limit` ， 控制多久为SSL连接做一次会话秘钥协商 (Magnus)

  可以设置为0来完全禁用重新协商，如果使用了一个破损的库的话是需要这样的。 特别的，一些供应商为CVE-2009-3555提供了紧急补丁，引起重新协商的尝试失败。

- 修复后端启动期间可能的死锁 (Tom)

- 修复由于relcache重新干净的加载期间没有处理错误引起的可能的崩溃 (Tom)

- 修复在子事务启动中尝试从一个错误中恢复时可能的崩溃 (Tom)

- 修复几个与使用保存点和客户端编码与服务器编码不同有关的内存泄露 (Tom)

- 修复GIST索引页分裂的最后恢复清理期间不正确的WAL数据发出 (Yoichi Hirai)

  这会导致索引损坏，更甚至可能在WAL重放期间的一个错误， 如果我们很不幸的在完成一个不完整的GIST插入之后最后恢复清理期间崩溃。

- 让 `substring()` 对待所有 `bit` 类型的负的长度为 "所有剩余的字符串" (Tom)

  以前的代码只以这种方式对待-1，并且会为其他负值产生一个无效的结果值， 可能导致崩溃 (CVE-2010-0442)。

- 修复整数到位字符串的转换，当输出位的宽度比给出的整数宽，不同于8位的倍数时， 正确的处理第一部分的字节 (Tom)

- 修复正则表达式匹配病理上缓慢的一些情况 (Tom)

- 修复后端历史文件中的 `STOP WAL LOCATION` 项， 当结束位置正好是一个段的边缘时，报告下一个WAL段的名字 (Itagaki Takahiro)
- 修复更多情况下临时文件的泄露 (Heikki)

  这纠正了一个在以前的小版本中引入的问题。 失败的一个情况是plpgsql函数的返回集在另一个函数的异常处理中调用时。
- 改善约束排除处理布尔变量的情况，特别的，让他有可能排除一个有 "bool_column = false"约束的分区 (Tom)
- 当读取 `pg_hba.conf` 和相关的文件时，如果 `@` 出现在双引号标记内部， 那么就不将 `@something` 看做一个文件包含请求；另外， 永不将 `@` 本身看做一个文件包含请求 (Tom)

  这阻止了角色或数据库名以 `@` 开头时的古怪行为。 如果你需要包括路径名包含空格的文件，你仍然可以这样做，但是必须写 `@"/path to/file"` 而不是让双引号包含整个构造。
- 如果一个路径被命名为 `pg_hba.conf` 和相关文件中的包含目标， 那么阻止某些平台上的无限循环 (Tom)
- 如果 `SSL_read` 或 `SSL_write` 没有设置 `errno` 而失败，则修复可能的无限循环 (Tom)

  据报道，这在openssl的某些Windows版本中是可能的。
- 修复psql的 `numericlocale` 选项， 不要格式化不应该是latex和troff输出格式的字符串 (Heikki)
- 当 `ON_ERROR_STOP` 和 `--single-transaction` 都指定了并且一个错误发生在隐含的 `COMMIT` 期间时， 让psql返回正确的退出状态(3) (Bruce)
- 修复一个复合字段设置为NULL情况下的plpgsql失败 (Tom)
- 修复从PL/PerlU调用PL/Perl函数或反过来时可能的失败 (Tim Bunce)
- 在PL/Python中添加 `volatile` 标记，避免可能的编译器具体的错误行为 (Zdenek Kotala)
- 确保PL/Tcl完全初始化Tcl解释器 (Tom)

  这个疏忽唯一已知的症状是如果使用Tcl 8.5或更高版本， Tcl `clock` 命令错误行为。
- 阻止太多的关键字段指定到一个 `dblink_build_sql_*` 函数时， `contrib/dblink` 中的崩溃 (Rushabh Lathia, Joe Conway)
- 修复由于粗心的内存管理引起的 `contrib/xml2` 中的各种崩溃 (Tom)
- 让 `contrib/xml2` 的建立在Windows上更加稳健 (Andrew)
- 修复Windows信号处理中的竞态条件 (Radu Ilie)

这个错误一个已知的症状是 `pg_listener` 中的行在重负载的情况下会被删除。

- 更新时区数据文件到tzdata版本2010e，因为DST规律在Bangladesh、Chile、Fiji、Mexico、Paraguay、Samoa发生了改变。

E.86. 版本 8.2.15

> 发布日期: 2009-12-14

这个版本包含各种自8.2.14以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.86.1. 迁移到版本 8.2.15

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.14的版本升级而来， 那么请查看8.2.14的版本声明。

E.86.2. 修改列表

- 防御由于索引函数改变会话本地状态引起的间接安全威胁 (Gurjeet Singh, Tom)

  这个修改阻止了据说不变的索引函数有可能破坏超级用户的会话 (CVE-2009-4136)。

- 拒绝在公共名（CN）字段中包含一个嵌入的空字节的SSL认证 (Magnus)

  这阻止了SSL校验期间证书到服务器或客户端名的无意的匹配 (CVE-2009-4034)。

- 修复后端启动时缓存初始化期间可能的崩溃 (Tom)

- 阻止信号在不安全的时间中断 `VACUUM` (Alvaro)

  这个修复阻止了 `VACUUM FULL` 在它早已提交了它的元组活动之后取消时的PANIC， 和规划 `VACUUM` 在截断表之后被打断时的瞬态错误。

- 修复由于哈希表大小计算中整数溢出引起的可能的崩溃 (Tom)

  这在哈希连接的结果的大小有非常大的规划器估计时可能发生。

- 修复 `inet` / `cidr` 比较中非常少见的崩溃 (Chris Mikkelson)

- 确保没有忽略被预备事务持有的共享元组级别锁 (Heikki)

- 修复用于在一个子事务中存取的游标的临时文件的过早删除 (Heikki)

- 为GiST索引页分裂修复不正确的逻辑，当分裂依赖于索引的非第一字段时 (Paul Ramsey)

- 如果回收或删除一个老的WAL文件在检查点的结束失败，那么不要错误输出 (Heikki)

最好将这个问题看做是不重要的，并且允许检查点完成。未来的检查点将重试移除。 这样的问题在正常的操作中不会出现，但是已经看到会由错误设计Windows杀毒和备份软件引起。

- 确保WAL文件不会在Windows上反复的归档 (Heikki)

  如果一些其他进程介入不再需要的文件的删除中，这会是另外一个可能会发生的现象。

- 修复PAM口令处理，使其更加强健 (Tom)

  都知道以前的代码在组合Linux `pam_krb5` PAM模块和微软活动目录作为域控制器时会失败。 可能在其他地方也会有问题，因为它关于PAM堆栈会传送来什么参数做了不正确的假设。

- 修复 `CREATE OR REPLACE FUNCTION` 期间所有权依赖关系的处理 (Tom)

- 修复从 `plperlu` 调用 `plperl` 或反过来调用时的bug (Tom)

  从内部函数出错退出会导致由于未能为外部函数重新选择正确的Perl解释器而崩溃。

- 修复PL/Perl函数重新定义了时会话寿命内存泄露 (Tom)

- 确保当通过设置返回PL/Perl函数返回时，Perl数组正确的转换到了 PostgreSQL数组 (Andrew Dunstan, Abhijit Menon-Sen)

  非设置返回函数早已正确的工作了。

- 修复PL/Python异常处理中罕见的崩溃 (Peter)

- 确保psql的flex模块是用正确的系统头文件定义编译的 (Tom)

  这修复了在 `--enable-largefile` 导致生成的代码中不兼容的变化的平台上的建立失败。

- 让主进程忽略连接请求包中的任何 `application_name` 参数， 以提高和未来libpq版本的兼容性 (Tom)

- 更新时区缩写文件，匹配当前的实际情况 (Joachim Wieland)

  这包括添加 `IDT` 和 `SGT` 到缺省的时区缩写设置。

- 更新时区数据文件到tzdata版本2009s， 因为DST规律在Antarctica、Argentina、Bangladesh, Fiji、Novokuznetsk、 Pakistan、Palestine、Samoa、Syria发生了变化；还为Hong Kong做了历史纠正。

E.87. 版本 8.2.14

> 发布日期: 2009-09-09

这个版本包含各种自8.2.13以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.87.1. 迁移到版本 8.2.14

运行8.2.X的用户不需要转储/恢复。不过，如果你在 `interval` 字段上有任何哈希索引， 你必须在升级到8.2.14之后 `REINDEX` 他们。 另外， 如果你是从一个早于8.2.11的版本升级而来，那么请查看8.2.11的版本声明。

E.87.2. 修改列表

- 在 `pg_start_backup()` 期间强制WAL段切换 (Heikki)

  这避免了可能导致基础备份不可用的极端情况。

- 不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` 在安全定义函数的内部 (Tom, Heikki)

  这包含了一个在以前的修补中漏掉的情况，以前的修补是不允许 `SET ROLE` 和 `SET SESSION AUTHORIZATION` 在安全定义函数的内部。 (See CVE-2007-6600)

- 让一个早已加载的可加载模块的 `LOAD` 到一个空操作 (Tom)

  以前， `LOAD` 会尝试卸载并重新加载该模块， 但是这是不安全的并且不是所有的都有用。

- 在LDAP认证期间不允许空的密码 (Magnus)

- 修复出现在外部级别聚集函数的参数中的子SELECT的处理 (Tom)

- 修复与从排序或物化规划节点抓取整行值相关的bug (Tom)

- 当有多于100个子句在AND或OR列表中时，恢复禁用部分索引和约束排除优化的规划器改变 (Tom)

- 为数据类型 `interval` 修复哈希计算 (Tom)

  这纠正了哈希连接在间隔值上的错误结果。也改变了哈希索引在间隔字段上的内容。 如果您有任何这样的索引，您必须在升级之后 `REINDEX` 它们。

- 将 `to_char(..., 'TH')` 看做带有 `'HH'` / `'HH12'` 的大写字母顺序的后缀 (Heikki)

  以前作为 `'th'` （小写）处理。

- 当 `_x_` 超过2百万并且正在使用整数日期时间时，为 `INTERVAL '``_x_` ms'修复溢出 (Alex Hunsaker)

- 修复点到线段间距离的计算 (Tom)

  这导致几个几何运算符不正确的结果。

- 修复 `money` 数据类型，在货币数量没有小数点的环境下工作，比如日本 (Itagaki Takahiro)

- 适当的圆整日期时间输入，像 `00:12:57.9999999999999999999999999999` (Tom)

- 修复GiST R-tree操作符类中页分裂点的可怜的选择 (Teodor)

- 避免输入值是（接近）排序的顺序时批量插入到GIN索引中的性能退化 (Tom)

- 在PL/pgSQL的某些文本中正确的强制NOT NULL域约束 (Tom)

- 修复plperl初始化中的可移植性问题 (Andrew Dunstan)

- 如果 `postgresql.conf` 是空的，修复pg_ctl， 不要进入一个无限循环 (Jeff Davis)

- 当一个键或值太长而不适合它的数据结构时，让 `contrib/hstore` 抛出一个错误而不是悄悄的截断它 (Andrew Gierth)

- 修复 `contrib/xml2` 的 `xslt_process()` ， 正确的处理参数的最大数量（20） (Tom)

- 提高libpq代码的鲁棒性，在 `COPY FROM STDIN` 期间从错误中恢复 (Tom)

- 避免包含冲突的readline和editline头文件当两个库都安装了时 (Zdenek Kotala)

- 更新时区数据文件到tzdata版本20091， 因为DST规律在Bangladesh、Egypt、Jordan、Pakistan、Argentina/San_Luis、 Cuba、Jordan (只是历史纠正)、Mauritius、Morocco、Palestine、Syria、 Tunisia发生了改变。

E.88. 版本 8.2.13

> 发布日期: 2009-03-16

这个版本包含各种自8.2.12以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.88.1. 迁移到版本 8.2.13

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.11的版本升级而来， 那么请查看8.2.11的版本声明。

E.88.2. 修改列表

- 当编码转换失败时，阻止错误的递归崩溃 (Tom)

  这个修改扩展了在最后两个小版本中为相关的失败情节所做的修复。 以前的修复严格限制在最初的问题报告，但是我们现在意识到编码转换函数抛出的 任何错误会潜在的在尝试报告错误时导致无限的递归。 因此解决方法是在我们发现我们进入一个递归错误报告情况时，禁用翻译和编码转换， 并为任何错误消息报告纯ASCII格式。 (CVE-2009-0922)

- 为指定的转换函数禁用带有错误编码的 `CREATE CONVERSION` (Heikki)

  这阻止了编码转换失败的一个可能的情节。以前的修改是为了防范相同地区的其他类型的失败。

- 修复给到 `to_char()` 的格式代码不适合数据参数的类型时的内核转储 (Tom)

- 修复C环境使用多字节编码时 `contrib/tsearch2` 中可能的失败 (Teodor)

  在 `wchar_t` 比 `int` 狭窄的平台上有可能会崩溃；尤其是Windows。

- 修复 `contrib/tsearch2` 分析器处理包含多个 `@` 字符的类似邮箱的字符串时的极端低效 (Heikki)

- 修复带有隐式强制的 `CASE WHEN` 的反编译 (Tom)

  当尝试检测或转储一个视图时，这个错误在启用断言的建立中会导致断言失败， 或其他情况下的一个"意外的CASE WHEN子句"错误消息。

- 修复可能的TOAST表的行类型所有者的错误分配 (Tom)

如果`CLUSTER`或`ALTER TABLE`的一个重写变体被除表所有者之外的一个人执行， 那么该表的TOAST表的`pg_type`项将被标记为这个人所有而结束。 这没有造成直接的问题，因为TOAST行类型上的权限不是被任何普通数据库操作检测的。 不过，如果一个人稍后尝试删除发出该命令的角色，那么它会导致意外的失败（在8.1或8.2中）， 或在这样做后来自pg_dump的"数据类型的所有者看起来无效" 警告（在8.3中）。

- 修复PL/pgSQL，不要将字符串中任何地方的`INSERT`之后的`INTO`看做INTO变量子句， 不只是在字符串开头的位置；特别的，不要在`CREATE RULE`中的`INSERT INTO`处失败 (Tom)

- 在块完全退出时清理PL/pgSQL的错误状态变量 (Ashesh Vashi 和 Dave Page)

  这对于PL/pgSQL本身来说不是什么问题， 但是该疏忽会引起PL/pgSQL调试器在检测一个函数的状态时崩溃。

- 在Windows上重试到`CallNamedPipe()`的失败的调用 (Steve Marshall, Magnus)

  看起来这个函数有时会有暂时性的失败；我们以前将任何失败看做系统错误， 这会混淆`LISTEN`/`NOTIFY`和其他操作。

- 添加`MUST` (Mauritius Island Summer Time) 到已知的时区缩写的缺省列表 (Xavier Bugaud)

E.89. 版本 8.2.12

> 发布日期: 2009-02-02

这个版本包含各种自8.2.11以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.89.1. 迁移到版本 8.2.12

运行8.2.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.2.11的版本升级而来， 那么请查看8.2.11的版本声明。

E.89.2. 修改列表

- 改善 `headline()` 函数中URL的处理 (Teodor)
- 改善 `headline()` 函数中超长标题的处理 (Teodor)
- 如果一个编码转换是用错误的编码转换函数为指定的编码对创建的， 那么阻止可能的断言失败或错误转换 (Tom, Heikki)
- 如果一个在PL/pgSQL中执行的语句被重写到另一个类型的语句，例如，如果一个 `INSERT` 被重写到一个 `UPDATE` ，那么修复可能的断言失败 (Heikki)
- 确保一个快照可以用于数据类型输入函数 (Tom)

  这主要影响用包含用户定义的稳定或不变的函数的 `CHECK` 约束声明的域。 这样的函数通常在没有设置快照时失败。
- 让使用了SPI的函数用于数据类型I/O中更加安全；特别的，用于域检查约束中 (Tom)
- 避免在 `VACUUM` 中不必要的锁定小表 (Heikki)
- 修复一个使得 `UPDATE RETURNING tableoid` 返回0而不是正确的OID的问题 (Tom)
- 修复传递平等应用到一个外连接子句时规划器对选择性的错误估计 (Tom)

  这会导致像 `... from a left join b on a.a1 = b.b1 where a.a1 = 42 ...` 这样的查询的不好的规划
- 改善优化器处理长的 `IN` 列表 (Tom)

  这个修改避免了在启用约束排除时在这样的列表上浪费大量的时间。

- 确保可持有游标的内容不依赖于TOAST表的内容 (Tom)

  以前，游标结果中大的字段值可能被表示为TOAST指针，如果引用表在该游标被读取之前被删除， 或者如果该大值被删除然后清理了，那么将会失败。普通游标不会发生这种情况， 但是在持有过去它创建的事务的游标中会发生。

- 修复设置返回函数还没有读取它的整个结果就被截断时的内存泄露 (Tom)
- 修复 `contrib/dblink` 的 `dblink_get_result(text,bool)` 函数 (Joe)
- 修复来自 `contrib/sslinfo` 函数的可能的垃圾输出 (Tom)
- 修复configure脚本，当不能为PL/Perl获取连接信息时，正确的报告失败 (Andrew)
- 让所有文档适当的引用 `pgsql-bugs` 和/或 `pgsql-hackers` ， 而不是现在退役的 `pgsql-ports` 和 `pgsql-patches` 邮件列表 (Tom)
- 更新时区数据文件到tzdata版本2009a（因为Kathmandu和Switzerland、 Cuba中的历史DST纠正）

E.90. 版本 8.2.11

> 发布日期: 2008-11-03

这个版本包含各种自8.2.10以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.90.1. 迁移到版本 8.2.11

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.7的版本升级而来， 那么请查看8.2.7的版本声明。另外，如果你正在运行一个早于8.2.X的版本， 建议在升级之后 `REINDEX` 所有GiST索引。

E.90.2. 修改列表

- 修复由于在一个删除之后标记错误的索引项"dead"引起的GiST索引损坏 (Teodor)

  这会导致索引搜索未能找到它们应该找到的行。损坏的索引可以使用 `REINDEX` 修复。

- 修复客户端编码不能表示本地化的错误消息时的后端崩溃 (Tom)

  之前我们解决过类似的问题，但是如果"character has no equivalent" 消息本身不能被转换的话它将仍然会失败。 该修复是在检测到这样的情况时禁用本地化并发送纯ASCII错误消息。

- 修复深层嵌套的函数在一个触发器中调用时，可能的崩溃 (Tom)

- 改善 `_expression_` `IN` (`_expression-list_`) 查询的优化 (Tom, per an idea from Robert Haas)

  查询变量在右侧的情况在8.2.X和8.3.X中的处理效率比在以前的版本中低。 该修复为这样的情况恢复了8.1的行为。

- 修复子 `SELECT` 出现在 `FROM` 中的函数调用、 一个多行 `VALUES` 列表或一个 `RETURNING` 列表中时规则查询的错误扩展 (Tom)

  这个问题通常的症状是一个"未识别的节点类型"错误。

- 修复重新扫描散列的聚合计划期间的内存泄露 (Neil)

- 确保当一个新定义的PL/pgSQL触发器被当做普通函数调用时报告一个错误 (Tom)

- 当使用 `ALTER SET TABLESPACE` 移动一个表到另一个表空间时， 阻止可能的 `relfilenode` 编号冲突 (Heikki)

  该命令尝试重新使用现有的文件名，而不是使用一个在目标目录中已知未使用的文件名。

- 修复单个查询条目匹配文本的第一个单词时，不正确的tsearch2标题生成 (Sushant Sinha)

- 修复在一个 `--enable-integer-datetimes` 建立中使用一个非ISO日期类型时， 间隔值中分数秒的不正确的显示 (Ron Mayer)

- 当传递的元组和元组描述符有不同的字段数时，确保 `SPI_getvalue` 和 `SPI_getbinval` 正确的行为 (Tom)

  当一个表有添加或删除的字段时，这个情况是正常的，但是这两个函数没有正确的处理。 唯一可能的后果是一个不正确的错误指示。

- 修复ecpg对 `CREATE ROLE` 的解析 (Michael)

- 修复 `pg_ctl restart` 最近的破损 (Tom)

- 确保 `pg_control` 是以二进制模式打开的 (Itagaki Takahiro)

  pg_controldata和pg_resetxlog 没有正确的做到这点，因此在Windows上可能会失败。

- 更新时区数据文件到tzdata版本2008i （因为DST规律在Argentina, Brazil, Mauritius, Syria发生了变化）

E.91. 版本 8.2.10

> 发布日期: 2008-09-22

这个版本包含各种自8.2.9以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.91.1. 迁移到版本 8.2.10

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.7的版本升级而来， 那么请查看8.2.7的版本声明。

E.91.2. 修改列表

- 修复btree WAL恢复代码中的bug (Heikki)

  如果WAL在页分裂操作的中途停止，则恢复失败。

- 修复潜在的 `datfrozenxid` 计算错误 (Alvaro)

  这个错误可能会解释一些最近未能删除老的 `pg_clog` 数据的报告。

- 扩大本地锁计数器从32到64位 (Tom)

  这个响应报告计数器在足够长的事务中可能会溢出，导致意外的"锁已经持有"错误。

- 修复在GiST索引扫描期间元组可能重复输出 (Teodor)

- 修复一个视图包含一个简单的 `UNION ALL` 构造时，丢失的权限检查 (Heikki)

  正确的检查了引用表的权限，但是视图本身的权限没有检查。

- 在执行器启动中添加检查，以确保 `INSERT` 或 `UPDATE` 产生的元组将匹配目标表的当前行类型 (Tom)

  `ALTER COLUMN TYPE`，跟着重新使用一个以前缓存的规划，会产生这种类型的情况。 该检查防卫了可能跟着发生的数据损坏和/或崩溃。

- 修复 `DROP OWNED` 期间可能的重复删除 (Tom)

  这通常会导致奇怪的错误，比如"关系NNN的缓存查找失败"。

- 修复 `AT TIME ZONE`，首先尝试解释它的时区参数为时区缩写， 并且只在失败时尝试它为一个全时区名，而不是像以前一样 (Tom)

时间戳输入函数总是以这个顺序解决模糊的时区名称。让 `AT TIME ZONE` 这样做也改善了一致性，并且修复了一个在8.1中引入的兼容性错误： 在模糊的情况下，我们现在和8.0和以前一样行为，因为在老的版本中， `AT TIME ZONE` 只接受缩写。

- 修复日期时间输入函数，当运行在一个64位的平台上时，正确的检测整数溢出 (Tom)
- 显示一个拥有单位的配置参数时，阻止单位转换期间的整数溢出 (Tom)
- 提高写入非常长的日志消息到系统日志的性能 (Tom)
- 在 `pg_hba.conf` 中允许空格在LDAP URL的前缀部分中 (Tom)
- 修复向后扫描 `SELECT DISTINCT ON` 查询上的游标中的错误 (Tom)
- 修复嵌套子查询表达式的规划器错误 (Tom)

  如果外部子查询没有直接的依赖于父查询，但是内部子查询直接依赖于父查询， 那么外部的值可能不会为新的父查询行计算。

- 修复规划器，估计 `GROUP BY` 表达式产生的布尔结果总是在两个组中， 不管表达式的内容是什么 (Tom)

  这非常明显的比为某些布尔测试，像 `_col_` `IS NULL`， 的普通 `GROUP BY` 估计更精确。

- 修复PL/pgSQL，当 `FOR` 循环的目标变量是一个包含复合类型字段的记录时不要失败 (Tom)
- 修复PL/Tcl，使其与Tcl8.5正确的行为，并且更加小心数据发送到或者来自Tcl的编码 (Tom)
- 在Windows上，通过阻止libpq尝试发送超过64kB的系统调用来绕开一个Microsoft错误 (Magnus)
- 改善未能发送一个SQL命令之后，pg_dump和pg_restore 的错误报告 (Tom)
- 修复pg_ctl，在 `restart` 时妥善保存主进程的命令行参数 (Bruce)
- 更新时区数据文件到tzdata版本2008f（因为DST规律在Argentina、Bahamas、 Brazil、Mauritius、Morocco、Pakistan、Palestine和Paraguay发生了改变）

E.92. 版本 8.2.9

> 发布日期: 2008-06-12

这个版本包含8.2.8的一个严重的和一个小的错误修复。关于8.2主版本的新特性信息，请查看Section E.101。

E.92.1. 迁移到版本 8.2.9

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.7的版本升级而来，那么请查看8.2.7的版本声明。

E.92.2. 修改列表

- 让 `pg_get_ruledef()` 给负的常量加上括号 (Tom)

  在该修复之前，视图或规则中负的常量可能被转储为，例如， `-42::integer` ， 这是微妙的不正确的：由于运算符优先级规则，它应该是 `(-42)::integer` 。 通常这将无关紧要，但是它会与另一个最近的补丁相互作用导致PostgreSQL 拒绝一个有效的 `SELECT DISTINCT` 视图查询。因为这会导致pg_dump 输出未能重载，所以它被看做一个高优先级修复。转储输出实际上不正确的唯一发布版本是8.3.1和8.2.7。

- 让 `ALTER AGGREGATE ... OWNER TO` 更新 `pg_shdepend` (Tom)

  如果该聚集稍后包含在一个 `DROP OWNED` 或 `REASSIGN OWNED` 操作中， 这个疏忽会导致问题。

E.93. 版本 8.2.8

> 发布日期: 从未发布

这个版本包含各种自8.2.7以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.93.1. 迁移到版本 8.2.8

运行8.2.X的用户不需要转储/恢复。不过，如果你是从一个早于8.2.7的版本升级而来， 那么请查看8.2.7的版本声明。

E.93.2. 修改列表

- 修复使用一个UTF-8数据库编码和一个不同的客户端编码时，发生在Windows上的 `ERRORDATA_STACK_SIZE exceeded` 崩溃 (Tom)
- 修复 `ALTER TABLE ADD COLUMN ... PRIMARY KEY`， 这样正确的检查新的字段看看是否初始化为所有非空 (Brendan Jurd)

  以前的版本完全疏忽了检查这个要求。
- 修复从继承自同一个祖先的约束的多个父关系中继承"相同的"约束时， 可能的 `CREATE TABLE` 失败 (Tom)
- 修复 `pg_get_ruledef()` 显示别名，如果有，附加 `UPDATE` 或 `DELETE` 的目标表 (Tom)
- 修复可能导致 `too many LWLocks taken` 失败的GIN错误 (Teodor)
- 避免反编译损坏的数据时可能的崩溃 (Zdenek Kotala)
- 修复两个地方，在这两个地方SIGTERM退出后端可能在共享内存中留下损坏的状态 (Tom)

  如果SIGTERM用于关闭整个数据库集群，那么哪种情况都不是非常重要的， 但是如果有人尝试SIGTERM单个后端，那么就有问题了。
- 修复ISO-8859-5和其他编码之间的转换，以处理Cyrillic "Yo"字符 (`e` 和 `E` 带有两个点） (Sergey Burladyan)
- 修复几个数据类型输入函数，尤其是 `array_in()`， 它们允许在它们的结果中未使用的字节包含未初始化的、不可预期的值 (Tom)

这会导致两个表面上相同的字面值被看做不等的失败，导致分析器抱怨未匹配的 `ORDER BY` 和 `DISTINCT` 表达式。

- 修复正则表达式子字符串匹配中的一个极端情况 （ `substring(``_string_` from `_pattern_`)） (Tom)

  这个问题出现在整体匹配但是用户已经指定了一个加上括号的子表达式， 并且该子表达式没有得到一个匹配的情况下。一个例子是 `substring('foo' from 'foo(bar)?')` 。 这应该返回NULL，因为 `(bar)` 没有匹配，但是它错误的返回了整个模式匹配 （也就是 `foo` ）。

- 更新时区数据文件到tzdata版本2008c（因为DST规律在Morocco、Iraq、 Choibalsan、Pakistan、Syria、Cuba和Argentina/San_Luis发生了改变）
- 修复ecpg的 `PGTYPEStimestamp_sub()` 函数的不正确结果 (Michael)
- 为 `contrib/tsearch2` 的 `tsquery` 类型修复破损的GiST比较函数 (Teodor)
- 修复 `contrib/cube` 函数中可能的崩溃 (Tom)
- 当输入查询返回一个NULL值时，修复 `contrib/xml2` 的 `xpath_table()` 函数中的内核转储 (Tom)
- 修复 `contrib/xml2` 的makefile，不要重载 `CFLAGS` (Tom)
- 修复 `DatumGetBool` 宏，使用gcc 4.3时不要失败 (Tom)

  这个问题影响返回布尔值的"老式的" (V0) C函数。该修复在8.3中早就有了， 但是向后修复的需要在当时没有意识到。

E.94. 版本 8.2.7

> 发布日期: 2008-03-17

这个版本包含各种自8.2.6以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.94.1. 迁移到版本 8.2.7

运行8.2.X的用户不需要转储/恢复。不过，如果你受下列描述的Windows环境问题影响， 你可能需要在升级之后 `REINDEX` 在文本字段上的索引。

E.94.2. 修改列表

- 为认为不同字符组合为相等的Windows环境修复字符串比较 (Tom)

  这个修复只应用于Windows并且是在使用UTF-8数据库编码时。 两年前为所有其他情况作了相同的修复，但是使用UTF-8的Windows使用一个单独的代码路径， 所以没有更新。如果你正在使用一个认为一些不相同的字符串相等的环境， 你可能需要 `REINDEX`，以修复文本字段上现存的索引。

- 修复在不同的系统目录上并发 `VACUUM FULL` 操作之间潜在的死锁 (Tom)
- 修复长期存在的 `LISTEN` / `NOTIFY` 竞态条件 (Tom)

  在稀有情况下，一个刚刚执行了 `LISTEN` 的会话可能不会获得一个通知， 即使预期应该有一个，因为观察到并发事务执行 `NOTIFY` 在稍后提交。

  该修复的一个副作用是一个已经执行了还未提交的 `LISTEN` 命令的事务对于该 `LISTEN` 将不会看到任何 `pg_listener` 中的行， 而它应该看到；以前它能看到。这个行为从未记录过，但是有可能有一些应用依赖于老的行为。

- 不允许 `LISTEN` 和 `UNLISTEN` 在一个预备事务中 (Tom)

  以前是允许的，但是尝试这样做会有各种不愉快的结果，尤其是只要 `UNLISTEN` 保持未提交，那么原始的后端就不会退出。

- 不允许在一个预备事务中删除一个临时表 (Heikki)

  这在8.1中正确的禁用了，但是在8.2中无意的打破了检查。

- 修复一个错误发生在查询使用哈希索引期间时的罕见崩溃 (Heikki)

- 修复某些设置返回函数使用中的内存泄露 (Neil)
- 修复公元前2月29的日期时间值的输入 (Tom)

  以前的代码关于哪年是闰年是错误的。
- 修复 `ALTER OWNER` 的一些变体中的"未识别的节点类型"错误 (Tom)
- 确保退出一个锁等待时， `pg_stat_activity`.`waiting` 标记被清除了 (Tom)
- 修复Windows Vista上进程权限的处理 (Dave, Magnus)

  特别的，这个修复允许作为管理员用户启动服务器。
- 更新时区数据文件到tzdata版本2008a（特别的，最近智利的变化）；调整时区缩写 `VET` (Venezuela)意为UTC-4:30，不是UTC-4:00 (Tom)
- 修复pg_ctl，正确的从命令行选项中提取主进程的端口号 (Itagaki Takahiro, Tom)

  以前， `pg_ctl start -w` 会在错误的端口尝试连接主进程，导致启动失败的错误报告。
- 使用 `-fwrapv` 来防御最近的gcc版本中可能的错误优化 (Tom)

  在使用gcc 4.3或更高版本建立PostgreSQL时，这是必要的。
- 正确的强制 `statement_timeout` 值比 `INT_MAX` 微妙（大约35分钟）更长 (Tom)

  这个错误只影响使用 `--enable-integer-datetimes` 的建立。
- 修复常量折叠简化一个子查询时，"意外的 PARAM_SUBLINK ID"规划器错误 (Tom)
- 修复约束排除处理 `IS NULL` 和 `NOT` 表达式中的逻辑错误 (Tom)

  规划器有时会由于NULL结果的可能性排除不应该被排除的分区。
- 修复另一个导致"未能建立一个 N-way 连接"规划器错误的原因 (Tom)

  这在利用一个连接子句之前必须强制一个缺乏子句的连接的情况下会发生。
- 修复外连接规划中不正确的常数传播 (Tom)

  规划器有时不正确的推断一个变量会被迫等于一个常量，导致错误的查询结果。
- 修复 `ORDER BY` 和 `GROUP BY` 中常量表达式的显示 (Tom)

  明确计算的常量会不正确的显示。这会导致，例如转储或重载期间视图定义的损坏。
- 修复libpq，在COPY OUT期间正确的处理NOTICE消息 (Tom)

  这个失败只有在一个用户定义数据类型的输出例程发出一个NOTICE时能观察到， 但是不保证它不会由于其他的原因而发生。

E.95. 版本 8.2.6

> 发布日期: 2008-01-07

这个版本包含各种自8.2.5以来的修复，包括重大安全问题的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.95.1. 迁移到版本 8.2.6

运行8.2.X的用户不需要转储/恢复。

E.95.2. 修改列表

- 阻止索引中的函数用用户运行 `VACUUM` 、 `ANALYZE` 等的权限执行 (Tom)

  索引表达式和局部索引谓词中使用的函数是在制作一个新的表条目时计算的。 一直以来大家都明白，如果一个人修改了一个不可信用户的表，那么会引起特洛伊木马执行的风险。 （注意，触发器、缺省、检查约束等，引起相同类型的风险。）但是索引中的函数会引起额外的风险， 因为它们将被日常维护操作执行，比如 `VACUUM FULL` ， 它们通常自动在超级用户账户下执行。 例如，一个不法用户可以通过设置一个特洛伊木马索引定义并等待下一个日常清理， 用超级用户的权限执行代码。该修复安排标准的维护操作（包括 `VACUUM` 、 `ANALYZE` 、 `REINDEX` 和 `CLUSTER` ）作为表所有者执行而不是作为调用用户执行， 使用早已为 `SECURITY DEFINER` 函数使用的相同的权限切换机制。 为了阻止绕开这个安全错误， `SET SESSION AUTHORIZATION` 和 `SET ROLE` 现在禁止在 `SECURITY DEFINER` 内容中执行。 (CVE-2007-6600)

- 修复正则表达式包中的各种错误 (Tom, Will Drewry)

  适当配置的正则表达式模式会导致崩溃、无限或接近无限循环和/或大量的内存消耗， 所有这些对于从受信任的源接受正则搜索模式的应用来说都会引起服务拒绝的危害。 (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- 要求使用 `/contrib/dblink` 的非超级用户只使用口令认证，作为一个安全措施 (Joe)

  出现在8.2.5中的该修复是不完整的，因为它只为一些 `dblink` 函数堵住了漏洞。 (CVE-2007-6601, CVE-2007-3278)

- 为GIN索引修复WAL重放中的错误 (Teodor)

- 修复 `maintenance_work_mem` 是4GB或更多时，GIN索引建立正确的工作 (Tom)

- 更新时区数据文件到tzdata版本2007k（特别的，最近的阿根廷的变化） (Tom)

- 改善规划器在非C环境中对LIKE/regex估计的处理 (Tom)
- 为深层外连接嵌套修复规划速度问题，和可能的连接顺序的贫乏的选择 (Tom)
- 修复 `WHERE false AND var IN (SELECT ...)` 的一些情况行中的规划器失败 (Tom)
- 让 `CREATE TABLE ... SERIAL` 和 `ALTER SEQUENCE ... OWNED BY` 不要改变序列的 `currval()` 状态 (Tom)
- 保存 `ALTER TABLE ... ALTER COLUMN TYPE` 重建的索引的表空间和存储参数 (Tom)
- 让归档恢复总是开始一个新的WAL时间线，而不是只在使用恢复停止时间时开始新的时间线 (Simon)

  这避免了尝试重写一个最后WAL段的现有归档拷贝的极端情况风险， 并且看起来比原先的定义更简单和干净。
- 让 `VACUUM` 在表非常小而没什么用处时，不使用所有的 `maintenance_work_mem` (Alvaro)
- 修复使用一个多字节数据库编码时， `translate()` 中潜在的崩溃 (Tom)
- 让 `corr()` 为负的关联值返回正确的结果 (Neil)
- 为超过68年的间隔修复 `extract(epoch from interval)` 中的溢出 (Tom)
- 修复PL/Perl，当在一个信任的函数中使用UTF-8正则表达式时不要失败 (Andrew)
- 修复PL/Perl，以处理平台的Perl定义类型 `bool` 为 `int` 而不是 `char` 的情况 (Tom)

  虽然理论上这可能在任何地方发生，但是没有标准的Perl建立这样做…… 直到Mac OS X 10.5。
- 修复PL/Python，使其与Python 2.5在64位的机器上正确的工作 (Marko Kreen)
- 修复PL/Python，在长的意外消息上不要崩溃 (Alvaro)
- 修复pg_dump，正确的处理缺省表达式与它们的父表不同的继承子表 (Tom)
- 修复 `PGPASSFILE` 引用的文件不是一个纯文件时libpq的崩溃 (Martin Pitt)
- ecpg解析器修复 (Michael)
- 让 `contrib/pgcrypto` 防御OpenSSL库在键长于128位时失败； 这至少是在一些Solaris版本上的情况 (Marko Kreen)
- 让 `contrib/tablefunc` 的 `crosstab()` 处理NULL rowid为它本身的类别， 而不是崩溃 (Joe)
- 修复 `tsvector` 和 `tsquery` 输出例程，以正确的逃逸反斜杠 (Teodor, Bruce)
- 修复 `to_tsvector()` 在大的输入字符串上的崩溃 (Teodor)

- 在重新生成 `configure` 脚本时，要求使用一个特定的Autoconf版本 (Peter)

  这只影响开发者和打包者。做这个修改是为了阻止意外的使用未经测试的Autoconf 和PostgreSQL版本的组合。如果你真的想要使用一个不同的 Autoconf版本，你可以删除版本校验，但是结果如何就是你的责任了。

- 更新 `gettimeofday` 配置检查，这样PostgreSQL 可以在更新的MinGW版本上建立 (Magnus)

E.96. 版本 8.2.5

> 发布日期: 2007-09-17

这个版本包含各种自8.2.4以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.96.1. 迁移到版本 8.2.5

运行8.2.X的用户不需要转储/恢复。

E.96.2. 修改列表

- 当一个事务插入行然后在接近相同表上的并发 `VACUUM` 的结尾退出时， 阻止索引损坏 (Tom)
- 为在域中包含域的情况修复 `ALTER DOMAIN ADD CONSTRAINT` (Tom)
- 让 `CREATE DOMAIN ... DEFAULT NULL` 正确的工作 (Tom)
- 修复一些外连接的规划器问题，尤其是对 `t1 LEFT JOIN t2 WHERE t2.col IS NULL` 的可怜的大小估计 (Tom)
- 允许 `interval` 数据类型接受只由毫秒和微妙组成的输入 (Neil)
- 在 `timestamp` 输入中允许时区名出现在年之前 (Tom)
- 修复 `/contrib/tsearch2` 使用的GIN索引 (Teodor)
- 加速rtree索引插入 (Teodor)
- 修复过度的记录SSL错误消息 (Tom)
- 修复日志，这样在使用系统日志进程时日志消息不会交错 (Andrew)
- 修复 `log_min_error_statement` 日志耗尽内存时的崩溃 (Tom)
- 修复一些外键极端情况的不正确的处理 (Tom)
- 修复 `stddev_pop(numeric)` 和 `var_pop(numeric)` (Tom)
- 阻止 `REINDEX` 和 `CLUSTER` 由于尝试处理其他会话的临时表而失败 (Alvaro)
- 更新时区数据规则，特别是新西兰即将到来的变化 (Tom)

- Windows套接字和信号灯的改变 (Magnus)
- 让 `pg_ctl -w` 在Windows服务器模式正确的工作 (Dave Page)
- 修复在Windows上使用MIT Kerberos时的内存分配错误 (Magnus)
- 抑制Windows上日志时间戳中的时区名(`%Z`)，由于可能的编码错误匹配 (Tom)
- 要求使用 `/contrib/dblink` 的非超级用户只使用口令认证，作为一个安全措施 (Joe)
- 因为安全原因，限制 `/contrib/pgstattuple` 函数为超级用户 (Tom)
- 不要让 `/contrib/intarray` 尝试使用它自己的GIN操作符类作为缺省 （这会在转储/恢复时引起问题） (Tom)

E.97. 版本 8.2.4

> 发布日期: 2007-04-23

这个版本包含各种自8.2.3以来的修复，包括一个安全修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.97.1. 迁移到版本 8.2.4

运行8.2.X的用户不需要转储/恢复。

E.97.2. 修改列表

- 支持在 `search_path` 中明确的布置临时表模式，并禁用函数和操作符搜索它 (Tom)

  这需要允许一个安全定义函数设置一个 `search_path` 的真正安全的值。 没有它，一个无特权的SQL用户可以使用临时对象用安全定义函数的权限执行代码(CVE-2007-2138)。 参阅 `CREATE FUNCTION` 获取更多信息。

- 通过在每个后端中强制重载，为Windows修复 `shared_preload_libraries` (Korry Douglas)
- 修复 `to_char()` ，这样它正确的大写/小写本地化的天或月名 (Pavel Stehule)
- `/contrib/tsearch2` 崩溃修复 (Teodor)
- 要求 `COMMIT PREPARED` 在事务预备的数据库中执行 (Heikki)
- 允许 `pg_dump` 在Windows上的二进制备份比两千兆字节大 (Magnus)
- 新增传统的(台湾)中文FAQ (Zhou Daojing)
- 阻止统计收集器太频繁的写入磁盘 (Tom)
- 修复 `VACUUM FULL` 如何处理 `UPDATE` 链中潜在的数据损坏错误 (Tom, Pavan Deolasee)
- 修复使用数组类型的域中的错误 (Tom)
- 修复 `pg_dump` ，这样它可以在不转储拥有的表时使用 `-t` 转储一个序列字段的序列 (Tom)
- 规划器修复，包括改善外连接和位图扫描选择逻辑 (Tom)

- 修复PL/pgSQL函数尝试从一个`EXCEPTION`块中`RETURN`时， 可能的错误恢复或崩溃 (Tom)
- 修复扩大哈希索引期间的PANIC (Tom)
- 修复POSIX风格的时区规格，以遵循新的USA DST规则 (Tom)

E.98. 版本 8.2.3

> 发布日期: 2007-02-07

这个版本包含两个8.2.2的修复。关于8.2主版本的新特性信息，请查看Section E.101。

E.98.1. 迁移到版本 8.2.3

运行8.2.X的用户不需要转储/恢复。

E.98.2. 修改列表

- 为约束和功能索引中的类型长度删除过度限制的检查 (Tom)
- 修复最优化，这样MIN/MAX在子查询中可以再次使用索引 (Tom)

E.99. 版本 8.2.2

> 发布日期: 2007-02-05

这个版本包含各种自8.2.1以来的修复，包括一个安全修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.99.1. 迁移到版本 8.2.2

运行8.2.X的用户不需要转储/恢复。

E.99.2. 修改列表

- 删除允许连接的用户读取后端内存的安全缺陷 (Tom)

  该缺陷包括抑制SQL函数返回它声明的数据类型和修改表字段的数据类型的正常检查 (CVE-2007-0555, CVE-2007-0556)。这些错误可以很容易的被利用来导致一个后端崩溃， 并且原理上可能被用来读取该用户不应该能够访问的数据库内容。

- 修复由于选择一个不可行的分裂点，btree索引页分裂可能失败的不那么罕见的错误 (Heikki Linnakangas)

- 修复Borland C编译脚本 (L Bayuk)

- 正确的处理以 `00` 结束的年的 `to_char('CC')` (Tom)

  2000年是二十世纪，而不是二十一世纪。

- `/contrib/tsearch2` 本地化改善 (Tatsuo, Teodor)

- 修复 `information_schema.key_column_usage` 视图中不正确的权限检查 (Tom)

  症状是"relation with OID nnnnn does not exist"错误。要不使用 `initdb` 来修复这个错误， 使用 `CREATE OR REPLACE VIEW` 安装在 `share/information_schema.sql` 中找到的正确的定义。 注意，你将需要在每个数据库中执行。

- 为拥有许多表的数据库提高 `VACUUM` 性能 (Tom)

- 修复由 `UNION` 触发的罕见的Assert()崩溃 (Tom)

- 修复使用 `ROW` 不平等条件的索引搜索中潜在的不正确的结果 (Tom)

- 为超过三字节长度的UTF8序列加强多字节字符处理的安全 (Tom)

- 修复由于尝试同步早已删除的文件，在Windows上发生的假的"没有权限"失败 (Magnus, Tom)
- 修复可能导致状态收集器在Windows上挂掉的错误 (Magnus)

  这会反过来导致自动清理不工作。
- 修复一个早已使用的PL/pgSQL函数被更新时可能的崩溃 (Tom)
- 改善PL/pgSQL对域类型的处理 (Sergiy Vyshnevetskiy, Tom)
- 修复处理PL/pgSQL异常块中可能的错误 (Tom)

E.100. 版本 8.2.1

> 发布日期: 2007-01-08

这个版本包含各种自8.2以来的修复。关于8.2主版本的新特性信息， 请查看Section E.101。

E.100.1. 迁移到版本 8.2.1

运行8.2.X的用户不需要转储/恢复。

E.100.2. 修改列表

- 修复 `SELECT` ... `LIMIT ALL` (还有 `LIMIT NULL`)的崩溃 (Tom)
- `Several /contrib/tsearch2` 修复 (Teodor)
- 在Windows上，让来自操作系统的日志消息使用ASCII编码 (Hiroshi Saito)

  这修复了操作系统和数据库服务器的编码错误匹配时的转换问题。
- 修复使用 `win32.mak` 的pg_dump的Windows连接 (Hiroshi Saito)
- 为外连接查询修复规划器错误 (Tom)
- 修复几个包括子SELECT的查询中的问题 (Tom)
- 修复子事务退出期间SPI中潜在的崩溃 (Tom)

  这影响所有PL函数，因为他们都使用SPI。
- 提高PDF文件的建立速度 (Peter)
- 重新添加JST (Japan)时区缩写 (Tom)
- 改善与索引扫描相关的最优化决策 (Tom)
- 让psql像以前一样输出多字节组合字符，而不是作为 `\u` 输出 (Tom)
- 改善使用括号的正则表达式的索引使用 (Tom)

  这也提高了psql `\d` 性能。
- 当从一个早于8.2的服务器中转储时，让pg_dumpall假设数据库有公共的 `CONNECT` 权限 (Tom)

  这保留了以前如果 `pg_hba.conf` 允许，那么任何人都可以连接到数据库的行为。

E.101. 版本 8.2

> 发布日期: 2006-12-05

E.101.1. 概述

这个版本添加了许多用户要求的功能和性能改善，包括：

- 查询语言增强包括 `INSERT/UPDATE/DELETE RETURNING` ， 多行 `VALUES` 列表和 `UPDATE` / `DELETE` 中可选的目标表别名。
- 不带有并发锁的索引创建 `INSERT` / `UPDATE` / `DELETE` 操作
- 许多查询最优化改善，包括对重新排序外连接的支持
- 提高降低内存使用的排序性能
- 带有更好的并发性的更多有效的锁定
- 更有效的清理
- 简化热备服务器的管理
- 为表和索引新增 `FILLFACTOR` 支持
- 监视、记录和性能调优的添加
- 对创建和删除对象的更多控制
- 表继承管理可以为已经存在的表定义或从已经存在的表中删除
- `COPY TO` 可以拷贝任意 `SELECT` 语句的输出
- 数组的改进，包括空数组
- 聚集函数的改进，包括多重输入聚集和SQL:2003统计函数
- 许多 `contrib/` 的改进

E.101.2. 迁移到版本 8.2

想要从任何以前的版本中迁移数据的用户需要使用pg_dump转储/恢复。

观察下列的不兼容性：

- 缺省设置 `escape_string_warning` 为 `on` (Bruce)

如果反斜杠逃逸在非逃逸(非 `E''`)字符串 中使用，这会发出一个警告。

- 修改行构造函数语法 (`ROW(...)`)， 这样列表元素 `foo.*` 将被扩展到一个他们的成员字段的列表， 而不是和以前一样创建一个嵌套的行类型字段 (Tom)

  新的行为实际上更有用，因为它允许，例如，触发器用 `IF row(new.*) IS DISTINCT FROM row(old.*)` 检查数据的改变。 通过省略 `.*` ，仍然可以使用老的行为。

- 让行比较遵循SQL 标准语义，并允许它们在索引扫描中使用 (Tom)

  以前，row = 和 <>比较遵循标准，但是< <= > >=不遵循标准。 行比较现在可以用作多字段索引的索引约束匹配行值。

- 让row `IS [NOT] NULL` 测试遵循SQL标准语义 (Tom)

  前者的行为在 `IS NULL` 的简单情况下符合标准，但是如果任意行字段是非空的， `IS NOT NULL` 将返回真，而标准认为只有所有字段都是非空的时才应该返回真。

- 让 `SET CONSTRAINT` 只影响一个约束 (Kris Jurka)

  在以前的版本中， `SET CONSTRAINT` 用一个匹配的名字修改所有的约束。 在这个版本中，模式搜索路径只用于修改第一个匹配的约束。 也支持模式声明。这更加符合SQL标准。

- 因为安全原因，为表删除 `RULE` 权限 (Tom)

  这个版本，只有表的所有者可以为该表创建或修改规则。为了向后兼容， 仍然接受 `GRANT` / `REVOKE RULE` ， 但是它什么也不做。

- 数组比较的改进 (Tom)

  现在也比较数组的维度。

- 修改数组连接，以匹配记录的行为 (Tom)

  这改变了串联会修改数组下界的以前的行为。

- 让postmaster和postgres 的命令行选项相同 (Peter)

  这允许主进程不使用 `-o` 就能传递参数到每个后端。请注意， 某些选项现在只能用作长格式的选项，因为与单字母的选项有冲突。

- 反对使用postmaster符号连接 (Peter)

  postmaster和postgres命令现在动作相同， 行为由命令行选项决定。为了兼容性，保留了postmaster符号连接， 但是实际上不需要了。

- 修改 `log_duration` ， 使其在查询没有输出的情况下也输出 (Tom)

  在以前的版本中， `log_duration` 只在查询出现在日志的前面时才输出。

- 让 `to_char(time)` 和 `to_char(interval)` 对待 `HH` 和 `HH12` 为12小时间隔

  大多数应用应该使用 `HH24`，除非它们想要12小时显示。

- 在从 `INET` 到 `CIDR` 的转换中有零个未标记的位 (Tom)

  这确保了转换的值对于 `CIDR` 实际有效。

- 删除 `australian_timezones` 配置变量 (Joachim Wieland)

  为了配置时区缩写，这个变量被一个更通用的设备取代。

- 为嵌套的循环索引扫描改善开销估计 (Tom)

  这可以消除设置不切实际的小的 `random_page_cost` 值的需要。如果你已经使用了一个非常小的 `random_page_cost`，请重新检查你的测试用例。

- 改变 `pg_dump` `-n` 和 `-t` 选项的行为。 (Greg Sabino Mullane)

  请查看 `pg_dump` 手册页获取细节。

- 修改libpq `PQdsplen()` 以返回一个有用的值 (Martijn van Oosterhout)

- 声明libpq `PQgetssl()` 返回 `void *`，而不是 `SSL *` (Martijn van Oosterhout)

  这允许应用不包括OpenSSL头使用函数。

- 为了版本兼容性检查，C语言可加载模块现在必须包括一个 `PG_MODULE_MAGIC` 宏调用 (Martijn van Oosterhout)

- 为了安全起见，PL/PerlU函数使用的模块不再可用于PL/Perl函数 (Andrew)

  > **Note:** 这也意味着数据在PL/Perl函数和PL/PerlU函数之间不再能够共享。 某些Perl安装没有用正确的允许多个解释器存在于一个进程中的标识编译。 在这种情况下，PL/Perl和PL/PerlU不能同时用于一个后端中。 解决办法是获得一个支持多个解释器的Perl安装。

- 在 `contrib/xml2/` 中，重命名 `xml_valid()` 为 `xml_is_well_formed()` (Tom)

  为了向后兼容，将保留 `xml_valid()`，但是它的行为将在未来的版本中改变为模式检查。

- 删除 `contrib/ora2pg/`，现在在http://www.samse.fr/GPL/ora2pg中

- 删除已经迁移到PgFoundry的贡献模块： `adddepend`，`dbase`，`dbmirror`，`fulltextindex`，`mac`，`userlock`

- 删除废弃的贡献模块： `mSQL-interface`，`tips`

- 删除QNX和BEOS端口 (Bruce)

  这些端口不再有活跃的维护者。

E.101.3. 修改列表

下面你将看到PostgreSQL 8.2和以前的主版本之间详细的变化。

E.101.3.1. 性能提升

- 允许规划器在某些情况下重新排序外连接 (Tom)

  在以前的版本中，外连接总是以在查询中写入的顺序计算。这个修改允许查询优化器考虑重新排序外连接， 在它确定改变连接顺序而不改变查询的意思的情况下。这对于包含多个外连接或混合内部和外部连接的查询来说， 会产生一个可观的性能差异。

- 提高 `IN` (表达式列表) 子句的效率 (Tom)
- 提高排序速度和减少内存使用 (Simon, Tom)
- 提高子事务性能 (Alvaro, Itagaki Takahiro, Tom)
- 添加 `FILLFACTOR` 到表和 索引创建(ITAGAKI Takahiro)

  这在每个表或索引页中留有额外的自由空间，允许随着数据的增长提高性能。 这对于维护集群来说特别有价值。

- 为 `shared_buffers` 和 `max_fsm_pages` 增加默认值 (Andrew)
- 通过打断锁管理器表为部分，提高锁的性能 (Tom)

  这允许锁定粒度更细，减少争用。

- 减少序列扫描的锁定需求 (Qingqing Zhou)
- 减少数据库创建和销毁的锁定需求 (Tom)
- 为 `LIKE`，`ILIKE` 和 正则表达式操作改善优化器的选择性估计。
- 改善连接的规划，以继承的表和 `UNION ALL` 视图 (Tom)
- 允许约束排除应用于 继承的 `UPDATE` 和 `DELETE` 查询 (Tom)

  `SELECT` 早已遵循约束排除。

- 改善常量 `WHERE` 子句的规划，比如一个只依赖于从一个外部查询级别继承的变量的条件 (Tom)
- 协议级别未命名的预备语句是为每个 `BIND` 值组重新规划的 (Tom)

  因为准确的参数值可以在计划中使用，所以提高了性能。

- 加速清理B-Tree索引 (Heikki Linnakangas, Tom)

- 避免在 `VACUUM` 期间额外的扫描没有索引的表 (Greg Stark)
- 提高多字段GiST索引 (Oleg, Teodor)
- 在B-Tree页分裂之前删除死的索引条目 (Junji Teramoto)

E.101.3.2. 服务器的变化

- 允许强制切换到一个新的事务日志文件 (Simon, Tom)

  这对于保持热备份从服务器与主服务器同步是有价值的。事务日志文件切换现在在 `pg_stop_backup()` 期间也自动发生。这确保了恢复所需要的所有事务日志文件可以立即归档。

- 添加WAL信息函数 (Simon)

  为从 `pg_stop_backup()` 和相关函数显示的十六进制WAL位置中询问当前事务日志插入点和决定 WAL文件名添加函数。

- 改善WAL重放期间从一个崩溃中恢复 (Simon)

  服务器现在在WAL恢复期间定期的检查点，所以如果有一个崩溃， 未来的WAL恢复被缩短。这样如果崩溃了， 就消除了对热备份服务器重放自基础备份以来的整个日志的需要。

- 改善长期的WAL重放的可靠性 (Heikki, Simon, Tom)

  以前，由于XID包围，向前推进超过20亿事务的尝试将不会工作。 这意味着热备服务器必须定期从干净的基础备份中重载。

- 添加 `archive_timeout`， 强制事务日志文件在给定的时间间隔切换 (Simon)

  这为热备份服务器执行最大的复制延迟。

- 添加本地LDAP认证 (Magnus Hagander)

  这对于不支持PAM的平台，比如Windows，尤其有用。

- 添加 `GRANT CONNECT ON DATABASE` (Gevik Babakhani)

  这给出了数据库访问的SQL级别控制。它作为一个额外的过滤器工作， 紧接着现有的 `pg_hba.conf` 控制。

- 添加对SSL证书撤销列表 (CRL) 文件的支持 (Libor Hohoš)

  服务器和libpq现在都能识别CRL文件。

- GiST索引现在是可集群的 (Teodor)

- 删除例程自动清理服务器日志条目 (Bruce)

  `pg_stat_activity` 现在显示自动清理活动。

- 在单独的表而不是整个数据库中跟踪最大的XID寿命 (Alvaro)

  这通过避免不必要的VACUUM,减少了阻止事务ID环绕中包括的开销。

- 添加最后的清理和分析时间戳字段到状态收集器 (Larry Rosenman)

  这些值现在出现在 `pg_stat_*_tables` 系统视图中。

- 提高统计监测的性能，尤其是 `stats_command_string` (Tom, Bruce)

  这个版本缺省启用 `stats_command_string` ，现在它的开销是最小的。 这意味着 `pg_stat_activity` 现在缺省将显示所有活动的查询。

- 添加一个 `waiting` 字段到 `pg_stat_activity` (Tom)

  这允许 `pg_stat_activity` 显示所有包括在ps显示中的信息。

- 添加配置参数 `update_process_title` ， 控制ps的显示是否为每个命令都更新了 (Bruce)

  在更新ps显示昂贵的平台上，有必要把这个关掉，完全依赖于 `pg_stat_activity` 显示状态信息。

- 允许在配置设置中指定单位 (Peter)

  例如，你现在可以设置 `shared_buffers` 为 `32MB`， 而不是在心里转换大小。

- 在 `postgresql.conf` 中添加对包含指令 的支持 (Joachim Wieland)

- 改善协议级别预备/绑定/执行消息的记录 (Bruce, Tom)

  这样的日志现在显示语句名字、绑定参数值和被执行查询的文本。 另外， 当通过 `log_min_error_statement` 启用时，查询文本正确的包含在记录的错误消息中。

- 阻止 `max_stack_depth` 被设置为不安全的值

  在我们可以决定实际内核堆栈深度限制的平台上（大多数是）， 确保 `max_stack_depth` 的初始缺省值是安全的，并拒绝尝试设置它为不安全的大值。

- 在更多情况下的查询中启用突出显示错误位置 (Tom)

  服务器现在能够为一些语义错误报告一个具体的错误位置（比如未识别的字段名）， 而不是和以前一样只报告基本的语法错误。

- 修复 `VACUUM` 中"未能重新找到父键"错误 (Tom)

- 在服务器重新启动期间清理 `pg_internal.init` 缓存文件 (Simon)

这避免了PITR恢复之后缓存文件可能包含陈旧数据的风险。

- 修复通过 `VACUUM` 截断十亿字节范围的大关系时的竞态条件 (Tom)
- 修复行级别锁上导致不需要的死锁错误的bug (Tom)
- 修复影响几十亿字节哈希索引的错误 (Tom)
- 每个后端进程现在是它自己的进程组领导 (Tom)

  这允许查询取消退出从一个后端或归档/恢复进程中调用的子进程。

E.101.3.3. 查询的变化

- 添加 `INSERT` / `UPDATE` / `DELETE` `RETURNING` (Jonah Harris, Tom)

  这允许这些命令返回值，比如为一个新行计算序列键。在 `UPDATE` 的情况下， 返回来自行的更新版本的值。

- 添加对多行 `VALUES` 子句的支持， 按照SQL标准 (Joe, Tom)

  这允许 `INSERT` 插入多行常量，或查询使用常量生成结果集。例如， `INSERT ... VALUES (...), (...), ....` , and `SELECT * FROM (VALUES (...), (...), ....) AS alias(f1, ...)` 。

- 允许 `UPDATE` 和 `DELETE` 为目标表使用一个别名 (Atsushi Ogawa)

  SQL标准不允许在它们的命令中使用别名，但是许多数据库系统为了标记方便，无论如何都允许一个。

- 允许 `UPDATE` 在一个值的列表中设置多个字段值 (Susanne Ebrecht)

  这基本上是一个按对分配字段和值的速记。语法是 `UPDATE tab SET (``_column_` , ...) = ( `_val_` , ...)。

- 使行比较按照标准工作 (Tom)

  格式<, <=, >, >=现在按字母顺序比较行，也就是，比较第一个元素， 如果相等则比较第二个元素，以此类推。以前它们在所有的元素上扩展成一个AND条件， 这既不是标准也不是非常有用。

- 添加 `CASCADE` 选项到 `TRUNCATE` (Joachim Wieland)

  这导致 `TRUNCATE` 自动包括通过外键引用指定的表的所有表。 虽然方便，但是这是一个危险的工具—小心的使用它！

- 在同一个 `SELECT` 命令中支持 `FOR UPDATE` 和 `FOR SHARE` (Tom)
- 添加 `IS NOT DISTINCT FROM` (Pavel Stehule)

这个操作符类似于等于(`=`)，但是当左侧和右侧的操作数都是 `NULL` 时， 评估为真，当只有一侧为 `NULL` 时评估为假，而不是在这些情况下生成 `NULL` 。

- 改善 `UNION` / `INTERSECT` / `EXCEPT` 使用的长度输出 (Tom)

  当所有对应的字段有相同的定义长度时，该长度用于结果，而不是一个通用的长度。

- 允许 `ILIKE` 为多字节编码工作 (Tom)

  内部的， `ILIKE` 现在调用 `lower()` 然后使用 `LIKE` 。 地区设定的正则表达式模式在这些编码中仍然不工作。

- 使 `standard_conforming_strings` 能变成 `on` (Kevin Grittner)

  这允许禁用字符串中的反斜杠逃逸，让PostgreSQL更加标准兼容。 为了向后兼容，缺省是 `off` ， 但是未来的版本将改成缺省为 `on` 。

- 不要在子查询目标列表中平面化包含 `volatile` 函数的子查询 (Jaime Casanova)

  这阻止了由于 `volatile` 函数（比如 `random()` 或 `nextval()` ） 的多次评估引起的意外行为。它可能导致在不需要标记为 `volatile` 的函数面前性能退化。

- 添加系统视图 `pg_prepared_statements` 和 `pg_cursors` ， 显示预备语句和打开的游标 (Joachim Wieland, Neil)

  这在池连接设置中非常有用。

- 在 `EXPLAIN` 和 `EXECUTE` 中支持入口参数 (Tom)

  这允许，例如，JDBC `?` 参数在这些命令中工作。

- 如果SQL级别的 `PREPARE` 参数没有指定，那么从查询的内容中推断它们的类型 (Neil)

  协议层 `PREPARE` 早已这样做了。

- 允许 `LIMIT` 和 `OFFSET` 超过二十亿 (Dhanaraj M)

E.101.3.4. 对象操作的变化

- 添加 `TABLESPACE` 子句到 `CREATE TABLE AS` (Neil)

  这允许为新表指定表空间。

- 添加 `ON COMMIT` 子句到 `CREATE TABLE AS` (Neil)

  这允许临时表在事务提交时被截断或删除。该表缺省的行为是保持到会话结束。

- 添加 `INCLUDING CONSTRAINTS` 到 `CREATE TABLE LIKE` (Greg Stark)

  这允许容易的拷贝 `CHECK` 约束到一个新表。

- 允许创建占位符（壳）类型 (Martijn van Oosterhout)

  一个壳类型声明创建一个类型名，没有声明该类型的任何细节。 这使得壳类型是有用的，因为它允许更干净的声明类型的输入/输出函数， 它们在该类型被定义为"真的"之前必须存在。语法是 `CREATE TYPE` `_typename_` 。

- 聚集函数 现在支持多个输入参数 (Sergey Koposov, Tom)

- 添加新的聚集创建语法 (Tom)

  新的语法是 `CREATE AGGREGATE` `_aggname_` (`_input_type_`) (`_parameter_list_`)。 更自然的支持新的多参数聚集功能。仍然支持以前的语法。

- 添加 `ALTER ROLE PASSWORD NULL` ， 删除以前设置的角色密码 (Peter)

- 为许多对象类型添加 `DROP` 对象 `IF EXISTS` (Andrew)

  这允许 `DROP` 操作一个不存在的对象时不会生成错误。

- 添加 `DROP OWNED` ， 删除一个角色拥有的所有对象 (Alvaro)

- 添加 `REASSIGN OWNED` ， 重新分配一个角色拥有的所有对象 (Alvaro)

  这个和上面的 `DROP OWNED` ，帮助删除角色。

- 添加 `GRANT ON SEQUENCE` 语法 (Bruce)

  这是为了设置特定于序列的权限添加的。为了后向兼容， 仍然支持序列的 `GRANT ON TABLE` 。

- 为序列添加 `USAGE` 权限， 只允许 `currval()` 和 `nextval()` ，不允许 `setval()` (Bruce)

  `USAGE` 权限允许更精细的控制序列访问。 `USAGE` 允许用户增加一个序列，但是阻止他们使用 `setval()` 设置该序列为任意值。

- 添加 `ALTER TABLE [ NO ] INHERIT` (Greg Stark)

  这允许继承动态调整，而不是只在表创建和销毁时。 这在使用继承实现表分区时是非常有价值的。

- 允许在全局对象上的注释全局存储 (Kris Jurka)

  以前，附加到数据库的注释存储在单独的数据库中，使得它们无效， 并且没有规定注释规则或表空间。这个修改添加了一个新的共享目录 `pg_shdescription` 并在数据库、角色和表空间上存储注释。

E.101.3.5. 工具命令的变化

- 添加选项以允许索引创建时不阻塞并发的写入到表 (Greg Stark, Tom)

新的语法是 `CREATE INDEX CONCURRENTLY` 。 缺省的行为在索引被创建时仍然阻塞表的修改。

- 提供咨询锁功能 (Abhijit Menon-Sen, Tom)

  这是一个新的锁API，设计来替换曾经在/contrib/userlock中的那个。 userlock代码现在在pgfoundry上。

- 允许 `COPY` 转储 `SELECT` 查询 (Zoltan Boszormenyi, Karel Zak)

  这允许 `COPY` 转储任意的SQL查询。 语法是 `COPY (SELECT ...) TO` 。

- 让 `COPY` 命令返回一个命令标签， 该标签包括拷贝的行数 (Volkan YAZICI)

- 允许 `VACUUM` 终止行， 而不受其他并发 `VACUUM` 操作的影响 (Hannu Krossing, Alvaro, Tom)

- 让initdb 检测操作系统环境并相应的设置缺省 `DateStyle` (Peter)

  这使得安装的 `postgresql.conf` `DateStyle` 值更有可能正是所需要的。

- 减少initdb显示的进展消息的数量 (Tom)

E.101.3.6. 日期/时间的变化

- 在 `timestamp` 的输入值中允许完整的时区名 (Joachim Wieland)

  例如， `'2006-05-24 21:11 America/New_York'::timestamptz` 。

- 支持可配置的时区缩写 (Joachim Wieland)

  所需的一组时区缩写可以通过配置参数 `timezone_abbreviations` 选择。

- 添加 `pg_timezone_abbrevs` 和 `pg_timezone_names` 视图， 显示支持的时区 (Magnus Hagander)

- 添加 `clock_timestamp()` 、 `statement_timestamp()` 和 `transaction_timestamp()` (Bruce)

  `clock_timestamp()` 是当前挂钟的时间， `statement_timestamp()` 是当前语句到达服务器的时间， `transaction_timestamp()` 是 `now()` 的一个别名。

- 允许 `to_char()` 打印本地化的月和日的名字 (Euler Taveira de Oliveira)

- 允许 `to_char(time)` 和 `to_char(interval)` 输出AM/PM规格 (Bruce)

  间隔和时间是看做24小时周期的，例如 `25 hours` 被认为是AM。

- 添加新的函数 `justify_interval()` ， 调整间隔单位 (Mark Dilger)

- 允许时区偏移距离GMT达到14:59

Kiribati使用GMT+14，所以我们最好接受它。

- 间隔计算的改进 (Michael Glaesemann, Bruce)

E.101.3.7. 其他数据类型和函数的变化

- 允许数组包含 NULL 元素 (Tom)
- 允许分配数组元素与现有的项不连续 (Tom)

  介于中间的数组位置将用null填充。这是符合SQL标准的。

- 为数组子集比较新增内建的操作符 (@> 、 <@ 、 &&) (Teodor, Tom)

  这些操作符可以被使用GiST或GIN 索引的许多数据类型索引。

- 在 INET / CIDR 值上添加方便的算术 操作 (Stephen R. van den Berg)

  新增的操作符是 & (and)、 | (or)、 ~ (not)、 inet + int8 、 inet - int8 和 inet - inet 。

- 从SQL:2003添加新的聚集函数 (Neil)

  新增的函数是 var_pop() 、 var_samp() 、 stddev_pop() 和 stddev_samp() 。 var_samp() 和 stddev_samp() 只是重命名现有的聚集 variance() 和 stddev() 。后者的名字为了后向兼容仍然可用。

- 添加SQL:2003统计聚集 (Sergey Koposov)

  新增函数： regr_intercept() 、 regr_slope() 、 regr_r2() 、 corr() 、 covar_samp() 、 covar_pop() 、 regr_avgx() 、 regr_avgy() 、 regr_sxy() 、 regr_sxx() 、 regr_syy() 、 regr_count() 。

- 允许domains基于其他的域 (Tom)
- 适当的加强域的 CHECK 约束 (Neil, Tom)

  例如，声明返回域类型的用户定义函数的结果现在检查该域的约束。 这关闭了域实现中的一个重大漏洞。

- 修复转储重命名的 SERIAL 字段的问题 (Tom)

  该修复是通过明确的指定序列的 DEFAULT 和序列元素转储一个 SERIAL ，并且在重新加载时使用一个新的 ALTER SEQUENCE OWNED BY 命令重新构造该 SERIAL 的字段。这也允许删除一个 SERIAL 字段声明。

- 添加一个服务器端休眠函数 pg_sleep() (Joachim Wieland)
- 为 tid (tuple id) 数据类型添加所有比较操作符 (Mark Kirkwood, Greg Stark, Tom)

E.101.3.8. PL/pgSQL服务器端语言的变化

- 添加 `TG_table_name` 和 `TG_table_schema` 到触发器参数 (Andrew)

  `TG_relname` 现在已经废弃了。也为其他PL在触发器参数中做了类型的变化。

- 允许 `FOR` 语句返回标量值、记录和行类型 (Pavel Stehule)
- 添加一个 `BY` 子句到 `FOR` 循环，控制迭代增量 (Jaime Casanova)
- 添加 `STRICT` 到 `SELECT INTO` (Matt Miller)

  为了Oracle PL/SQL兼容性，如果多于或少于一行被 `SELECT` 返回， 则 `STRICT` 模式抛出一个异常。

E.101.3.9. PL/Perl服务器端语言的变化

- 添加 `table_name` 和 `table_schema` 到触发器参数 (Adam Sjøgren)
- 添加预备查询 (Dmitry Karasik)
- 让 `$_TD` 触发一个全局变量的数据 (Andrew)

  在以前，这是一个词汇，引起意外的分享违规行为。

- 为了安全起见，在单独的解释器中运行PL/Perl和PL/PerlU (Andrew)

  在序列中，它们再也不能分享数据和加载模块。另外， 如果Perl没有用必要的标识编译以允许多个解释器，那么在任何给出的后端进程中， 只可以使用这些语言中的一种。

E.101.3.10. PL/Python服务器端语言的变化

- 命名的参数是作为普通变量传送的，和在 `args[]` 数组中一样 (Sven Suursoho)
- 添加 `table_name` 和 `table_schema` 到触发器参数 (Andrew)
- 允许返回复合类型和结果集 (Sven Suursoho)
- 作为 `list` 、 `iterator` 或 `generator` 返回结果集 (Sven Suursoho)
- 允许函数返回 `void` (Neil)
- 现在支持Python 2.5了 (Tom)

E.101.3.11. psql 的变化

- 添加新的命令 `\password` ，用客户端侧的口令加密修改角色口令 (Peter)

- 允许 `\c` 连接到一个新的主机和端口号 (David, Volkan YAZICI)
- 添加表空间显示到 `\l+` (Philip Yarra)
- 改善 `\df` 斜线命令，包含函数的参数名和模式 (`OUT` 或 `INOUT`) (David Fetter)
- 支持二进制 `COPY` (Andreas Pflug)
- 添加选项在一个事务中运行整个会话 (Simon)

  使用选项 `-1` 或 `--single-transaction` 。
- 支持使用一个游标按批次的自动检索 `SELECT` 结果 (Chris Mair)

  这是使用 `\set FETCH_COUNT` `_n_` 启用的。 这个特性允许在psql中检索大的结果集， 而不用尝试在内存中缓存整个结果集。
- 让多行值对齐在适当的字段内 (Martijn van Oosterhout)

  包含换行符的字段值现在以更可读的方式显示。
- 多行语句保存为一个条目，而不是一次一行 (Sergey E. Koposov)

  这使得向上箭头调回查询更加简单。（在Windows上是不可用的， 因为该平台使用操作系统中的本地命令行编辑。）
- 让行计数器是64位的，这样它可以处理行数超过二十亿的文件 (David Fetter)
- 为 `INSERT` / `UPDATE` / `DELETE RETURNING` 报告返回的数据库和命令状态标签 (Tom)

E.101.3.12. pg_dump 的变化

- 允许对象的综合选择被pg_dump包括或排除 (Greg Sabino Mullane)

  pg_dump现在支持多个 `-n` （模式）和 `-t` （表）选项，并且添加了 `-N` 和 `-T` 选项排除对象。 另外，这些开关的参数现在可以是通配符表达式，而不是单个对象名， 例如 `-t 'foo*'` ，并且一个模式可以是 `-t` 或 `-T` 开关的一部分，比如 `-t schema1.table1` 。
- 添加pg_restore `--no-data-for-failed-tables` 选项，如果表创建失败则阻止加载数据（也就是，该表早已存在） (Martin Pitt)
- 添加pg_restore选项， 在单个事务中运行整个会话 (Simon)

  使用选项 `-1` 或 `--single-transaction` 。

E.101.3.13. libpq 的变化

- 添加 `PQencryptPassword()` ， 用来加密口令 (Tom)

这允许口令对于像 `ALTER ROLE ... PASSWORD` 这样的命令是预先加密发送的。

- 添加函数 `PQisthreadsafe()` (Bruce)

  这允许应用查询该库的线程安装状态。

- 添加 `PQdescribePrepared()` 、 `PQdescribePortal()` 和相关的函数返回关于以前预备好的语句和打开的游标的信息 (Volkan YAZICI)
- 允许LDAP从 `pg_service.conf` 中查找 (Laurenz Albe)
- 允许 `~/.pgpass` 中的主机名匹配缺省的套接字目录 (Bruce)

  一个空的主机名继续匹配任何Unix套接字连接， 但是这个添加允许记录特定于机器上的几个主进程之一。

E.101.3.14. ecpg 的变化

- 允许 `SHOW` 将它的结果放入一个变量中 (Joachim Wieland)
- 添加 `COPY TO STDOUT` (Joachim Wieland)
- 添加回归测试 (Joachim Wieland, Michael)
- 主要源代码清理 (Joachim Wieland, Michael)

E.101.3.15. Windows 端口

- 允许MSVC编译PostgreSQL服务器 (Magnus, Hiroshi Saito)
- 为实用命令和pg_dump 添加MSVC支持 (Hiroshi Saito)
- 为Windows代码页 `1253` 、 `1254` 、 `1255` 和 `1257` 添加支持 (Kris Jurka)
- 在启动时删除权限， 这样服务器可以从一个管理员账号启动 (Magnus)
- 稳定性修复 (Qingqing Zhou, Magnus)
- 添加本地信号灯实现 (Qingqing Zhou)

  以前的代码模仿SysV信号灯。

E.101.3.16. 源代码的变化

- 添加GIN (Generalized Inverted iNdex) 索引访问方法 (Teodor, Oleg)
- 删除R-tree索引 (Tom)

Rtree已经使用GiST重新实现了。 在其他差异中，这意味着rtree索引现在支持通过预写式日志(WAL)崩溃恢复。

- 减少库不必要的连接到后端 (Martijn van Oosterhout, Tom)
- 添加一个配置标志，允许libedit优先于GNU readline (Bruce)

  使用配置 `--with-libedit-preferred` 。

- 允许安装到包含空格的目录 (Peter)
- 提高重新定位安装目录的能力 (Tom)
- 为使用Solaris编译器的Solaris x86_64 添加支持 (Pierre Girard, Theo Schlossnagle, Bruce)
- 添加DTrace支持 (Robert Lor)
- 添加 `PG_VERSION_NUM` ， 供想要使用>和< 比较测试后端C语言版本第三方应用程序使用 (Bruce)
- 添加 `XLOG_BLCKSZ` ， 独立于 `BLCKSZ` (Mark Wong)
- 添加 `LWLOCK_STATS` 定义，报告锁定活动 (Tom)
- 为未知的configure选项发出警告 (Martijn van Oosterhout)
- 为"plugin"库添加服务器支持，可以用于附加任务，比如调试和性能测量 (Korry Douglas)

  这由两个功能组成：一个允许单独加载的共享库沟通的"会合变量"的表， 和一个允许库被加载到特定会话而不明确与客户端应用合作的新的配置参数 `local_preload_libraries` 。 这允许外部插件实现像PL/pgSQL调试器这样的特性。

- 重命名现有的配置参数 `preload_libraries` 为 `shared_preload_libraries` (Tom)

  这样做是为了清楚的和 `local_preload_libraries` 比较。

- 添加新的配置参数 `server_version_num` (Greg Sabino Mullane)

  这类似于 `server_version` ， 但是这是一个整数，例如 `80200` 。 这允许应用更容易的做版本检查。

- 添加配置参数 `seq_page_cost` (Tom)
- 重新实现回归测试脚本为C程序 (Magnus, Tom)
- 允许可加载的模块分配共享内存和轻量级锁 (Marc Munro)
- 添加动态加载库的自动初始化和终结 (Ralf Engelschall, Tom)

如果库定义了这样的符号，则新增的函数 `_PG_init()` 和 `_PG_fini()` 被调用。 因此我们不再需要在 `shared_preload_libraries` 中指定一个初始化函数；我们可以假设库使用了 `_PG_init()` 约定。

- 添加 `PG_MODULE_MAGIC` 标题块到所有共享的对象文件 (Martijn van Oosterhout)

  该魔法模块阻止可加载对象文件和服务器间的版本错误匹配。

- 为AIX添加共享的库支持 (Laurenz Albe)
- 新增XML文档章节 (Bruce)

E.101.3.17. 贡献包的变化

- 主要tsearch2的改进 (Oleg, Teodor)
  - 多字节编码支持，包括UTF8
  - 查询重写的支持
  - 改进排序功能
  - 同义词字典支持
  - Ispell字典现在识别MySpell格式，OpenOffice使用它。
  - GIN支持
- 添加包含Pgadmin管理功能的adminpack模块 (Dave)

  这些函数提供了没有在缺省的PostgreSQL 服务器中出现的额外的文件系统访问例程。

- 添加sslinfo模块 (Victor Wagner)

  报告当前连接的SSL证书的信息。

- 添加pgrowlocks模块 (Tatsuo)

  这为特定的表显示了行锁信息。

- 添加hstore模块 (Oleg, Teodor)
- 添加isn模块，替换isbn_issn (Jeremy Kronuz)

  这个新的实现支持EAN13、UPC、 ISBN (books)、ISMN (music)和ISSN (serials)。

- 添加索引信息函数到pgstattuple (ITAGAKI Takahiro, Satoshi Nagayasu)
- 添加pg_freespacemap模块，显示自由空间映射信息 (Mark Kirkwood)
- pgcrypto现在拥有所有计划的功能 (Marko Kreen)

- 在pgcrypto中包含iMath库，让公共秘钥加密函数总是可用。
- 添加在OpenBSD代码中丢失的SHA224算法。
- 在老的OpenSSL上为SHA224/256/384/512哈希激活内建代码，让这些算法总是可用。
- 新增函数gen_random_bytes()，返回密码强壮的随机性。对于生成加密秘钥是有用的。
- 删除digest_exists()、hmac_exists()和cipher_exists()函数。

- 改进多维数据集模块 (Joshua Reich)

  新增的函数是 `cube(float[])` 、 `cube(float[], float[])` 和 `cube_subset(cube, int4[])` 。

- 添加异步查询能力到dblink (Kai Londenberg, Joe Conway)
- 为数组子集比较新增操作符(@> 、 <@ 、 &&) (Tom)

  各种贡献包已经为它们的数据类型有了这些操作符，但是命名并不一致。 我们现在添加了一致命名的数组子集比较操作符到内核代码和所有拥有这样功能的贡献包。 （老的名字保留有效，但是已经废弃了。）

- 为所有拥有安装脚本的贡献包添加卸载脚本 (David, Josh Drake)

E.102. 版本 8.1.23

> 发布日期: 2010-12-16

这个版本包含各种自8.1.22以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

这预计是8.1.X系列的最后一个PostgreSQL版本。 推荐用户尽快更新到一个新的版本分支。

E.102.1. 迁移到版本 8.1.23

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.18的版本升级而来， 那么请查看8.1.18的版本声明。

E.102.2. 修改列表

- 强制Linux上 `wal_sync_method` 的缺省为 `fdatasync` (Tom Lane, Marti Raudsepp)

  Linux上的缺省实际上是 `fdatasync` 已经很多年了，但是最近的内核改变导致PostgreSQL选择了 `open_datasync` 。这个选择不会导致任何性能提升，并且在某些文件系统上引起了彻底的失败，尤其是 `ext4` 和 `data=journal` 挂载选项。

- 当起始检查点WAL记录和它的重做点不在同一个WAL段时，修复来自基础备份的恢复 (Jeff Davis)

- 添加在 `IA64` 上检测注册堆栈溢出的支持 (Tom Lane)

  `IA64` 体系结构有两个硬件堆栈。全面预防堆栈溢出失败需要两个都检查。

- 添加对 `copyObject()` 中堆栈溢出的检查 (Tom Lane)

  某些的代码路径可能会由于堆栈溢出给出一个足够复杂的查询而崩溃。

- 修复临时GiST索引中的页分裂检测 (Heikki Linnakangas)

  在一个临时索引上有一个"并发的"页分裂是可能的，如果例如当完成一个插入时正好有一个开放的游标扫描索引。GiST未能检测这种情况，并且因此导致游标的执行继续时给出错误的结果。

- 当正在 `ANALYZE` 复杂索引表达式时，避免内存泄露 (Tom Lane)

- 确保使用整行变量的索引仍然依赖于它的表 (Tom Lane)

像 `create index i on t (foo(t.*))` 这样的索引声明在它的表被删除时，将不会自动被删除。

- 不要用多个 `OUT` 参数"内联"一个SQL函数 (Tom Lane)

  这避免了由于丢失预期的结果行类型的信息而引起的可能的崩溃。

- 修复 `COALESCE()` 表达式的常量合并 (Tom Lane)

  规划器有时尝试评估子表达式而实际上是不可能的，有可能导致意想不到的错误。

- 为 `InhRelation` 结点添加打印功能 (Tom Lane)

  这避免了启用 `debug_print_parse` 和执行某些类型的查询时的失败。

- 修复点到水平线段间距离的不正确的计算 (Tom Lane)

  这个错误影响几个不同的几何距离测量操作符。

- 修复PL/pgSQL处理"简单的"表达式，在递归或错误恢复的情况下不要失败 (Tom Lane)

- 修复 `contrib/cube` 的GiST picksplit算法中的错误 (Alexander Korotkov)

  这会导致 `cube` 字段上GiST索引中大量的低效，和不完全正确的答案。如果你有这样的一个索引，考虑安装这个更新之后 `REINDEX` 它。

- 不要在 `contrib/dblink` 中发出"标识符将被截断"的通知，除非是在创建新的连接 (Itagaki Takahiro)

- 修复 `contrib/pgcrypto` 中丢失的公共键上潜在的内核转储 (Marti Raudsepp)

- 修复 `contrib/xml2` 的XPath查询函数中的内存泄露 (Tom Lane)

- 更新时区数据文件到tzdata版本2010o，因为DST规律在Fiji和Samoa发生了改变；还有对Hong Kong的历史纠正。

E.103. 版本 8.1.22

> 发布日期: 2010-10-04

这个版本包含各种自8.1.21以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

PostgreSQL社区将在2010年11月份停止对8.1.X版本系列的更新。建议用户尽快升级到新的版本分支。

E.103.1. 迁移到版本 8.1.22

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.18的版本升级而来， 那么请查看8.1.18的版本声明。

E.103.2. 修改列表

- 在PL/Perl和PL/Tcl中为每个调用的SQL用户id使用一个单独的解释器 (Tom Lane)

  这个修改阻止了破坏稍后将在同一个会话中在另一个SQL用户身份下执行的Perl或Tcl代码引起的安全问题（例如，在一个 `SECURITY DEFINER` 函数中）。大多数脚本语言提供很多可能执行的方式，比如重新定义被目标函数调用的标准函数或操作符。如果没有这个修改，任何拥有Perl或Tcl语言使用权限的SQL用户基本上都可以以目标函数所有者的SQL权限做任何事情。

  这个修改的代价是Perl和Tcl函数之间有意的沟通变得更加困难。为了提供一个逃逸出口，PL/PerlU和PL/TclU函数继续每个会话只使用一个解释器。这不认为是一个安全问题，因为所有这样的函数都在数据库超级用户的信任级别执行。

  有可能第三方程序语言声称提供受信任的执行有相似的安全问题。我们建议为了安全鉴定的目的联系任何你依赖的PL的作者。

  我们感谢Tim Bunce指出这个问题 (CVE-2010-3433)。

- 阻止 `pg_get_expr()` 中可能的崩溃，通过不允许它被一个参数调用，该参数不是它打算使用的系统目录字段之一 (Heikki Linnakangas, Tom Lane)

- 修复"不能处理未规划的子查询"错误 (Tom Lane)

  这在子查询包含一个连接别名引用，该引用扩展成一个包含另一个子查询的表达式时发生。

- 阻止show_session_authorization()在自动清理进程中崩溃 (Tom Lane)
- 防止函数返回行类型不是完全相同的记录集 (Tom Lane)
- 当哈希一个通过引用传递的函数结果时，修复可能的失败 (Tao Ma, Tom Lane)
- 当在写入锁文件时，小心的同步它们的内容（`postmaster.pid` 和套接字锁文件） (Tom Lane)

  如果机器在主进程启动之后很快就崩溃，那么这个疏忽会导致损坏锁文件的内容。转而阻止随后的尝试成功的启动主进程，直到手动移除锁文件。
- 当分配XID到深层嵌套的子事务时，避免递归 (Andres Freund, Robert Haas)

  如果限制了堆栈空间，那么原始的代码会导致一个崩溃。
- 修复 `log_line_prefix` 的 `%i` 逃逸，它会导致在后台启动时就产生垃圾 (Tom Lane)
- 修复启用归档时， `ALTER TABLE ... SET TABLESPACE` 中可能的数据损坏 (Jeff Davis)
- 允许 `CREATE DATABASE` 和 `ALTER DATABASE ... SET TABLESPACE` 被查询取消中断 (Guillaume Lelarge)
- 在PL/Python中，防止 `PyCObject_AsVoidPtr` 和 `PyCObject_FromVoidPtr` 产生空指针结果 (Peter Eisentraut)
- 改善 `contrib/dblink` 对包含删除字段的表的处理 (Tom Lane)
- 修复 `contrib/dblink` 中"重复的连接名"错误之后的连接泄露 (Itagaki Takahiro)
- 修复 `contrib/dblink` ，正确的处理连接名字长于62字节 (Itagaki Takahiro)
- 更新建立的基础结构和文档，以反映源代码仓库从CVS搬到了Git (Magnus Hagander and others)
- 更新时区数据文件到tzdata版本20101，因为DST规律在Egypt和Paletine发生了改变；还有对Finland的历史纠正。

  这个修改还为两个Micronesian时区添加了新的名字：Pacific/Chuuk现在比Pacific/Truk更受欢迎（并且首选的缩写是CHUT，不是TRUT），Pacific/Pohnpei比Pacific/Ponape更受欢迎。

E.104. 版本 8.1.21

> 发布日期: 2010-05-17

这个版本包含各种自8.1.20以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.104.1. 迁移到版本 8.1.21

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.18的版本升级而来， 那么请查看8.1.18的版本声明。

E.104.2. 修改列表

- 使用一个开放标记应用到整个解释器强制 `plperl` 中的限制，而不是使用 `Safe.pm` (Tim Bunce, Andrew Dunstan)

  最近的发展已经向我们证实：依赖于 `Safe.pm` 使得 `plperl` 可信任太不安全了。这个修改也一起删除了对 `Safe.pm` 的使用，为了支持使用一个总是应用开放代码标记的单独的解释器。这个修改令人愉快的副作用包括：现在有可能在 `plperl` 中以自然的方式使用Perl的 `strict` 编译，并且Perl的 `$a` 和 `$b` 变量在排序例程中像预期的那样工作，并且函数的编译显著的更快了。(CVE-2010-1169)

- 阻止PL/Tcl执行来自 `pltcl_modules` 的不受信任的代码 (Tom)

  PL/Tcl自动从数据库表中加载Tcl代码的特性可能会被特洛伊木马攻击利用，因为没有限制谁可以创建或插入到那个表。这个修改禁用了该特性，除非 `pltcl_modules` 是被超级用户拥有。（不过，该表上的权限是不检查的，所以实际上需要较小安全模块的表的安装仍然可以赋予合适的权限到受信任的非超级用户。）另外，阻止加载代码到不受限制的"普通"Tcl解释器中，除非我们真的要执行一个 `pltclu` 函数。 (CVE-2010-1170)

- 不要允许一个非特权用户重置超级用户仅有的参数设置 (Alvaro)

  以前，如果一个非特权用户为自己运行 `ALTER USER ... RESET ALL` ，或为他拥有的数据库运行 `ALTER DATABASE ... RESET ALL` ，将为该用户或数据库删除所有特殊的参数设置，甚至只应该是超级用户可改的参数。现在， `ALTER` 将只删除该用户有权限修改的参数。

- 如果关闭发生在 `CONTEXT` 添加到日志项时，避免后端关闭期间可能的崩溃 (Tom)

  在某些情况下，上下文打印函数将失败，因为当它要输出一个日志信息时当前事务早已回滚。

- 为现代的Perl版本更新pl/perl的 `ppport.h` (Andrew)
- 修复pl/python中的各种内存泄露 (Andreas Freund, Tom)
- 当扩展一个引用本身的变量时，阻止psql中的无限递归 (Tom)
- 确保 `contrib/pgstattuple` 函数迅速的响应取消中断 (Tatsuhito Kasahara)
- 让服务器启动正确的处理 `shmget()` 为一个现有的共享内存段返回 `EINVAL` 的情况 (Tom)

  这个行为在BSD驱动的内核上，包括OS X，已经被观察到了。它导致一个完全误导的启动失败抱怨共享内存请求尺寸太大。

- 更新时区数据文件到tzdata版本2010j，因为DST规律在Argentina、Australian Antarctic、Bangladesh、Mexico、Morocco、Pakistan、Palestine、Russia、Syria、Tunisia已经发生了改变；还为Taiwan做了历史纠正。

E.105. 版本 8.1.20

> 发布日期: 2010-03-15

这个版本包含各种自8.1.19以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.105.1. 迁移到版本 8.1.20

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.18的版本升级而来， 那么请查看8.1.18的版本声明。

E.105.2. 修改列表

- 添加新的配置参数 `ssl_renegotiation_limit`， 控制多久为一个SSL连接做一次会话秘钥协商 (Magnus)

  该参数可以设置为0来完全禁止重新协商，如果使用了一个破碎的SSL库可能需要这样。特别的， 一些供应商为CVE-2009-3555做了紧急补丁， 导致重新协商的尝试失败。

- 修复尝试从一个子事务启动失败中恢复时可能的崩溃 (Tom)

- 修复与使用保存点和客户端编码与服务器编码不同有关的服务器内存泄露 (Tom)

- 让 `substring()` 的 `bit` 类型认为任何负的长度意为"所有剩余的字符串" (Tom)

  以前的编码只对待-1以这个方式，并且会为其他负值产生一个无效的结果值，可能导致崩溃 (CVE-2010-0442)。

- 修复整数到位字符串的转换，当输出位宽度比给定的整数宽时（不同于8位的倍数）， 正确的处理第一个部分的字节 (Tom)

- 修复某些情况下病理上慢的正则表达式匹配 (Tom)

- 修复后端历史文件中的 `STOP WAL LOCATION` 入口，当结束位置正好在一个段的边界时报告下一个WAL段的名字 (Itagaki Takahiro)

- 修复更多情况下临时文件的泄露 (Heikki)

  这纠正了一个在以前的小版本中引入的问题。失败的一个案例是：当plpgsql函数返回集在另一个函数的异常处理中调用时。

- 当读取 `pg_hba.conf` 和相关的文件时，如果 `@` 出现在双引号标记的内部，不要将 `@something` 看做文件包含请求；另外，永远不要将 `@` 本身看做一个文件包含请求 (Tom)

  这阻止了角色或数据库名字以 `@` 开始时的不规律行为。如果你需要包含一个路径名包含空格的文件，你可以这样做，但是你必须写 `@"/path to/file"` 而不是让双引号包含整个结构。

- 如果一个路径以 `pg_hba.conf` 和相关文件中的包含目标命名，那么阻止某些平台上的无限循环 (Tom)
- 修复psql的 `numericlocale` 选项，不要格式化它不应该以latex和troff输出格式的字符串 (Heikki)
- 修复plpgsql在复合字段设置为NULL时的失败 (Tom)
- 在PL/Python中添加 `volatile` 标记，以避免可能的编译器特定不良行为 (Zdenek Kotala)
- 确保PL/Tcl完全初始化Tcl解释器 (Tom)

  这个疏忽唯一已知的症状是：如果使用Tcl 8.5或更新，Tcl `clock` 命令错误行为。

- 当太多的关键字段指定到一个 `dblink_build_sql_*` 函数时，阻止 `contrib/dblink` 中的崩溃 (Rushabh Lathia, Joe Conway)
- 修复粗心的内存管理引起的 `contrib/xml2` 中的各种崩溃 (Tom)
- 更新时区数据文件到tzdata版本2010e，因为DST规律在Bangladesh、Chile、Fiji、Mexico、Paraguay、Samoa发生了改变。

E.106. 版本 8.1.19

> 发布日期: 2009-12-14

这个版本包含各种自8.1.18以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.106.1. 迁移到版本 8.1.19

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.18的版本升级而来， 那么请查看8.1.18的版本声明。

E.106.2. 修改列表

- 防御由索引函数更改会话本地状态引起的间接安全威胁 (Gurjeet Singh, Tom)

  这个修改阻止了据称是不变的索引函数可能破坏一个超级用户的会话 (CVE-2009-4136)。

- 拒绝SSL认证在公共名（CN）字段包含一个嵌入的空字节 (Magnus)

  这阻止了SSL验证期间无意识的匹配证书到服务器或客户端名字 (CVE-2009-4034)。

- 修复后端启动缓存初始化期间可能的崩溃 (Tom)

- 阻止信号在不安全的时间中断 `VACUUM` (Alvaro)

  这个修改阻止了 `VACUUM FULL` 在它早已提交它的元组动作之后取消时的PANIC，还有如果计划 `VACUUM` 在截断表之后中断的瞬态错误。

- 修复由于哈希表尺寸计算中整数溢出而引起的可能的崩溃 (Tom)

  这在规划器估算哈希连接的结果非常大时会发生。

- 修复 `inet` / `cidr` 比较中非常罕见的崩溃 (Chris Mikkelson)

- 确保被预备事务持有的共享的元组级别锁不被忽略 (Heikki)

- 修复过早删除在一个子事务中访问的被游标使用的临时文件 (Heikki)

- 修复PAM口令处理，使其更加稳健 (Tom)

已知以前的代码未能组合Linux `pam_krb5` PAM模板和Microsoft Active Directory 作为域控制器。它可能在其他地方也有问题，因为它对于要传递哪个PAM堆栈参数做了不公正的假设。

- 修复 `CREATE OR REPLACE FUNCTION` 期间的所有权依赖关系的处理 (Tom)
- 当通过一个设置/返回PL/Perl函数返回时，确保Perl数组正确的转换到 PostgreSQL数组 (Andrew Dunstan, Abhijit Menon-Sen)

  这对于非设置返回函数来说早已正确的工作了。

- 修复PL/Python异常处理中罕见的崩溃 (Peter)
- 确保psql的flex模块是用正确的系统头定义编译的 (Tom)

  这修复了 `--enable-largefile` 导致在生成的代码中不兼容的改变的平台上的建立失败。

- 让主进程忽略任何在连接请求包中的 `application_name` 参数， 以改善与未来libpq版本的兼容性 (Tom)
- 更新时区数据文件到tzdata版本2009s， 因为DST规律已经在Antarctica、Argentina、Bangladesh、Fiji、Novokuznetsk、 Pakistan、Palestine、Samoa、Syria发生了改变；还为Hong Kong做了历史纠正。

E.107. 版本 8.1.18

> 发布日期: 2009-09-09

这个版本包含各种自8.1.17以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.107.1. 迁移到版本 8.1.18

运行8.1.X的用户不需要转储/恢复。不过，如果你在 `interval` 字段上有任何哈希索引，你必须在升级到8.1.18之后 `REINDEX` 它们。 另外，如果你是从一个早于8.1.15的版本升级而来，那么请查看8.1.15的版本声明。

E.107.2. 修改列表

- 不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` 在安全定义函数内部 (Tom, Heikki)

  这包含了在以前的补丁中丢失的一个情况，不允许 `SET ROLE` 和 `SET SESSION AUTHORIZATION` 在安全定义函数内部。 (请参阅 CVE-2007-6600)

- 修复出现在一个外部级别聚集函数的参数中的子查询的处理 (Tom)

- 为数据类型 `interval` 修复哈希计算 (Tom)

  这纠正了哈希连接在间隔值上的错误结果。也改变了哈希索引在间隔字段上的内容。 如果你有任何这样的索引，你必须在更新之后 `REINDEX` 它们。

- 将 `to_char(..., 'TH')` 看做大写的带有 `'HH'` / `'HH12'` 的序号前缀 (Heikki)

  以前是作为 `'th'` （小写） 处理的。

- 修复 `_x_` 超过2百万并且正在使用整数日期时间时， `INTERVAL '``_x_` ms'的溢出 (Alex Hunsaker)

- 修复点和线段之间距离的计算 (Tom)

  这导致一些几何操作符不正确的结果。

- 修复 `money` 数据类型在货币数量没有小数点的环境（比如日本）下工作 (Itagaki Takahiro)

- 适当的圆整像 `00:12:57.9999999999999999999999999999` 这样的日期时间输入 (Tom)

- 修复GiST R-tree操作符类中可怜的页分裂点的选择 (Teodor)

- 修复plperl初始化中的可移植性问题 (Andrew Dunstan)
- 修复pg_ctl，如果 `postgresql.conf` 为空时， 不要进入无限循环 (Jeff Davis)
- 修复 `contrib/xml2` 的 `xslt_process()` ， 正确的处理参数的最大数量（20） (Tom)
- 改善libpq代码的鲁棒性，从 `COPY FROM STDIN` 期间的错误中恢复 (Tom)
- 当两个库都安装了时，避免包含冲突的readline和editline头文件 (Zdenek Kotala)
- 更新时区数据文件到tzdata版本20091，因为DST规律在 Bangladesh、Egypt, Jordan、Pakistan、Argentina/San_Luis, Cuba、 Jordan(只是历史纠正)、Mauritius、Morocco、Palestine、Syria, Tunisia。

E.108. 版本 8.1.17

> 发布日期: 2009-03-16

这个版本包含各种自8.1.16以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.108.1. 迁移到版本 8.1.17

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.15的版本升级而来，那么请查看8.1.15的版本声明。

E.108.2. 修改列表

- 当编码转换失败时，阻止错误递归崩溃 (Tom)

  这个修改扩展了在最后两个小版本中对相关失败情节的修复。 以前的修复严格限制在原始的问题报告，但是我们现在意识到被编码转换函数抛出的 任何错误都潜在的导致在尝试报告该错误时无限的递归。 因此如果我们发现我们已经进入了一个递归错误报告的情节中， 解决方法是禁用翻译和编码转换，并且报告任何错误消息的纯ASCII格式。 (CVE-2009-0922)

- 不允许 `CREATE CONVERSION` 为指定的转换函数使用错误的编码 (Heikki)

  这阻止了编码转换失败的一种可能的情况。以前的修改是防范了相同区域中的其他类型的失败。

- 修复给定 `to_char()` 的格式化代码不适合数据参数的类型时的内核转储 (Tom)

- 修复 `CASE WHEN` 带有隐式强制时的反编译 (Tom)

  在尝试检查或转储一个视图时，这个错误在启用断言的建立中会导致断言失败， 或在其他情况下的一个"意外的CASE WHEN子句"错误消息。

- 修复TOAST表行类型的所有者的可能的错误分配 (Tom)

  如果 `CLUSTER` 或者 `ALTER TABLE` 的一个重写变体被表所有者之外的其他人执行， 该表的TOAST表的 `pg_type` 项将被标记为属于这个其他人而结束。 这没有造成直接问题，因为TOAST行类型的权限不被任何普通数据库操作检查。 不过，如果一个人稍后尝试删除发出该命令的角色（在8.1或8.2中）， 它会导致意外的失败， 或者pg_dump稍后也这样做时， 导致"数据类型的所有者看起来无效" 的警告（在8.3中）。

- 在块退出时完全清理PL/pgSQL的错误状态变量 (Ashesh Vashi and Dave Page)

  这对于PL/pgSQL本身来说不是一个问题，但是当检测一个函数的状态时，疏忽会导致PL/pgSQL调试器崩溃。

- 添加`MUST` (Mauritius Island Summer Time)到已知时区缩写的缺省列表 (Xavier Bugaud)

E.109. 版本 8.1.16

> 发布日期: 2009-02-02

这个版本包含各种自8.1.15以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.109.1. 迁移到版本 8.1.16

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.15的版本升级而来，那么请查看8.1.15的版本声明。

E.109.2. 修改列表

- 修复自动清理中的崩溃 (Alvaro)

  崩溃只在为反向事务包括的目的清理整个数据库时崩溃， 这意味着它发生的不频繁和难以追踪到。

- 改善 `headline()` 函数中URL的处理 (Teodor)
- 改善 `headline()` 函数中超长标题的处理 (Teodor)
- 如果编码转换是用错误的转换函数为指定的编码对创建的， 那么阻止可能的断言失败或错误转换 (Tom, Heikki)
- 避免 `VACUUM` 中小表的不必要的锁 (Heikki)
- 确保可持有游标的内容不依赖于TOAST表的内容 (Tom)

  以前，游标结果中大的字段值可能被表示为TOAST指针，如果引用表在该游标被读取之前被删除， 或者如果大的值被删除然后清理了，那么这就会失败。这在普通游标中不会发生， 但是会发生在保留了过去的创建事务的游标中。

- 修复 `contrib/tsearch2` 的 `get_covers()` 函数中未初始化的变量 (Teodor)
- 修复不能为PL/Perl获取连接信息时configure脚本适当的报告失败 (Andrew)
- 让所有文档适当的引用 `pgsql-bugs` 和/或 `pgsql-hackers` ， 替代现在退役的 `pgsql-ports` 和 `pgsql-patches` 邮件列表 (Tom)
- 更新时区数据文件到tzdata版本2009a （因为Kathmandu和在Switzerland、Cuba历史的DST纠正）

E.110. 版本 8.1.5

> 发布日期: 2008-11-03

这个版本包含各种自8.1.14以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.110.1. 迁移到版本 8.1.15

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。另外，如果你正在运行一个以前的8.1.X版本， 那么建议在升级之后 `REINDEX` 所有GiST索引。

E.110.2. 修改列表

- 修复由于在删除之后标记错误的索引项"dead"引起的GiST索引崩溃 (Teodor)

  这会导致索引搜索未能找到它们应该能找到的行。损坏的索引可以使用 `REINDEX` 修复。

- 修复客户端编码不能表示本地化错误消息时的后端崩溃(Tom)

  我们以前处理过相似的问题，但是如果"character has no equivalent" 消息本身不能被转换，那么它仍将会失败。修复是在我们检测到这样一个情况时， 禁用本地化和发送纯ASCII错误消息。

- 当深层嵌套的函数在一个触发器中调用时，修复可能的崩溃 (Tom)

- 当一个子 `SELECT` 出现在 `FROM` 、多行 `VALUES` 表或 `RETURNING` 列表中的函数调用中时，修复规则查询的错误膨胀 (Tom)

  这个问题的通常症状是一个"未识别的节点类型"错误。

- 当新定义的PL/pgSQL触发器函数被作为普通函数调用时，确保报告一个错误 (Tom)

- 当使用 `ALTER SET TABLESPACE` 移动一个表到另一个表空间时， 阻止可能的 `relfilenode` 编号冲突 (Heikki)

  该命令尝试重新使用现有的文件名字，而不是选择一个已知在目标目录中没有使用的文件名。

- 当单个查询条目匹配文本的第一个单词时，修复不正确的tsearch2标题生成 (Sushant Sinha)

- 当在 `--enable-integer-datetimes` 建立中使用了一个非ISO日期类型时， 修复间隔值中分数秒的不正确的显示 (Ron Mayer)
- 当传递的元组和元组描述符有不同的字段编号时，确保 `SPI_getvalue` 和 `SPI_getbinval` 正确的行为 (Tom)

  当表添加或删除了行时，这个情况是正常的，但是这两个函数没有适当的处理它。 唯一可能的后果是一个不正确的错误指示。
- 修复ecpg对 `CREATE ROLE` 的解析 (Michael)
- 修复 `pg_ctl restart` 最近的损坏 (Tom)
- 更新时区数据文件到tzdata版本2008i （因为DST规律在Argentina、Brazil, Mauritius, Syria发生了改变）

E.111. 版本 8.1.14

> 发布日期: 2008-09-22

这个版本包含各种自8.1.13以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.111.1. 迁移到版本 8.1.14

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.111.2. 修改列表

- 放宽本地锁计数器从32到64位 (Tom)

  这个响应报告了计数器会在足够长的事务中溢出，导致意外的"锁已被持有"错误。

- 修复GiST索引扫描期间可能的元组重复输出 (Teodor)

- 在执行器启动时添加检查，确保 `INSERT` 或 `UPDATE` 产生的元组将匹配目标表的当前行类型 (Tom)

  `ALTER COLUMN TYPE`， 跟着以前缓存的计划的重新使用， 会产生这种类型的情形。检查保护了可能接着发生的数据损坏和/或崩溃。

- 修复 `AT TIME ZONE`， 首先尝试解释他的时区参数为时区缩写， 如果失败，那么尝试它为完整时区名字，而不是和以前一样的其他绕开方式 (Tom)

  时间戳输入函数总是以这个顺序解决模糊的时区名字。让 `AT TIME ZONE` 也这样做提高了一致性，并且修复了一个在8.1中引进的兼容性bug： 在模糊情况下，我们现在的行为和8.0及以前的行为一致， 因为更老版本的 `AT TIME ZONE` 只接受缩写。

- 修复日期时间输入函数，以在64位平台上运行时能正确的检测到整数溢出 (Tom)

- 提高写入非常长的日志消息到系统日志的性能 (Tom)

- 修复 `SELECT DISTINCT ON` 查询上的游标向后扫描中的错误 (Tom)

- 修复嵌套子查询表达式的规划器bug (Tom)

  如果外侧子查询与父查询没有直接依赖关系，但是内侧子查询有， 那么外侧的值可能不会为新的父查询行重新计算。

- 修复规划器估计`GROUP BY`表达式产生的布尔结果总是在两个组中， 不管表达式的内容是什么 (Tom)

  对于某些布尔测试像`_col_ IS NULL`来说， 这比普通`GROUP BY`估计明显更加准确。

- 当`FOR`循环的目标变量是一个包含复合类型字段的记录时， 修复PL/pgSQL以不失败 (Tom)

- 修复PL/Tcl以与Tcl 8.5正确的行为，并且更加小心发送到或者来自Tcl的数据的编码 (Tom)

- 修复PL/Python以与Python 2.5一起工作

  这是一个8.2开发周期期间所做的修复的移植。

- 改善未能发送一个SQL命令之后，pg_dump和 pg_restore的错误报告 (Tom)

- 修复pg_ctl跨过一个`restart`时， 适当的保存主进程命令行参数 (Bruce)

- 更新时区数据文件的到tzdata版本2008f （因为DST规律在Argentina、Bahamas、Brazil、Mauritius, Morocco、Pakistan、 Palestine和Paraguay发生了改变）

E.112. 版本 8.1.13

> 发布日期: 2008-06-12

这个版本包含一个严重的和一个小的针对8.1.22的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.112.1. 迁移到版本 8.1.13

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.112.2. 修改列表

- 让 `pg_get_ruledef()` 给负的常量加上括号 (Tom)

  在该修复之前，在视图或规则中的负的常量可能被作为，例如 `-42::integer` 转储，而这是不正确的：它应该为 `(-42)::integer` ，因为操作符优先级规则。 通常这会导致小的差异，但是它会与另外一个最近的补丁相互作用导致PostgreSQL 拒绝一个有效的 `SELECT DISTINCT` 视图查询。因为这会导致pg_dump 输出不能加载，所以它被看做一个高优先级修复。实际上转储输出不正确的发行版本是8.3.1和8.2.7。

- 让 `ALTER AGGREGATE ... OWNER TO` 更新 `pg_shdepend` (Tom)

  如果聚集稍后包含在一个 `DROP OWNED` 或 `REASSIGN OWNED` 操作中， 那么这个疏忽会导致问题。

E.113. 版本 8.1.12

> 发布日期: 从未发布

这个版本包含各种自8.1.11以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.113.1. 迁移到版本 8.1.12

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.113.2. 修改列表

- 修复 `ALTER TABLE ADD COLUMN ... PRIMARY KEY`， 这样新的字段正确的检查是否它被初始化为所有都是非空 (Brendan Jurd)

  以前的版本完全忽略了检查这个要求。

- 修复从多个继承自同一个祖先的约束的父关系中继承"相同的"约束时， 可能的 `CREATE TABLE` 失败 (Tom)

- 修复ISO-8859-5和其他编码之间的转换，以处理Cyrillic "Yo"字符 （ `e` 和 `E` 带有两个句点） (Sergey Burladyan)

- 修复一个新的日期类型输入函数， 允许未使用的字节在它们的结果中包含未初始化的、不可预测的值 (Tom)

  这会导致两个表面上相同的文字值不被看做相等的失败， 导致解析器抱怨不匹配的 `ORDER BY` 和 `DISTINCT` 表达式。

- 修复正则表达式子串匹配中的一个极端情况(`substring(``_string_` from `_pattern_`)) (Tom)

  当有一个到模式的完全匹配，但是用户已经指定了一个加上括号的子表达式， 并且该子表达式没有获得一个匹配时会发生这个问题。一个例子是 `substring('foo' from 'foo(bar)?')` 。这个应该返回NULL， 因为 `(bar)` 没有匹配，但是它错误的返回了整个模式匹配 （也就是 `foo` ）。

- 更新时区数据文件到tzdata版本2008c （因为DST规律在Morocco、Iraq、Choibalsan、Pakistan、Syria、Cuba、 Argentina/San_Luis和Chile发生了改变）

- 修复ecpg的 `PGTYPEStimestamp_sub()` 函数中的不正确的结果 (Michael)
- 当输入查询返回一个NULL值时，修复 `contrib/xml2` 的 `xpath_table()` 函数中的内核转储 (Tom)
- 修复 `contrib/xml2` 的makefile，不要覆盖 `CFLAGS` (Tom)
- 修复 `DatumGetBool` 宏，不要在使用gcc 4.3时失败 (Tom)

  这个问题影响返回布尔的"老式的" (V0) C函数。 这个修复在8.3中已经有了，但是后向修复它的需要在当时没有意识到。

- 修复长期存在的 `LISTEN` / `NOTIFY` 竞态条件 (Tom)

  在罕见的情况下，刚刚执行了 `LISTEN` 的会话可能不会获得一个通知， 即使预期应该有一个通知，因为并发事务执行 `NOTIFY` 是在提交之后能观察到。

  该修复的一个副作用是一个刚刚执行了暂未提交的 `LISTEN` 命令的事务将不会看到该 `LISTEN` 的 `pg_listener` 中的任何行，而它应该能看到的； 以前它是能够看到的。这个行为不管怎样都没有记录过，但是有可能一些应用依赖于老的行为。

- 不允许 `LISTEN` 和 `UNLISTEN` 在一个准备事务中 (Tom)

  这在以前是允许的，但是尝试这样做会有各种不愉快的后果， 尤其是只要 `UNLISTEN` 保持未提交，原始的后端就不能退出。

- 修复在查询使用哈希索引期间发生错误时的罕见的崩溃 (Heikki)
- 修复公元前的年中二月29的日期时间值的输入 (Tom)

  以前的代码弄错了哪一年是闰年。

- 修复在某些 `ALTER OWNER` 的变体中的"未识别的节点类型"错误 (Tom)
- 修复pg_ctl，正确的从命令行选项中提取主进程的端口号 (Itagaki Takahiro, Tom)

  以前， `pg_ctl start -w` 尝试在错误的端口连接主进程， 导致启动失败的虚假的报告。

- 使用 `-fwrapv` 防卫在最近的gcc版本中可能的错误最优化 (Tom)

  这在用gcc 4.3或更新的版本建立PostgreSQL时是必需的。

- 修复 `ORDER BY` 和 `GROUP BY` 中常量表达式的显示 (Tom)

  一个显示转换的常量将会不正确的显示。这会导致例如转储和重载期间视图定义的损坏。

- 修复libpq，以在COPY OUT期间正确的处理NOTICE消息 (Tom)

只有当用户定义的数据类型的输出例程发出一个NOTICE时，这个失败才能观察到， 但是不保证它不会因为其他原因发生。

E.114. 版本 8.1.11

> 发布日期: 2008-01-07

这个版本包含各种自8.1.10以来的修复，包括对重要安全问题的修复。 关于8.1主版本的新特性信息，请查看Section E.125。

这是PostgreSQL社区为Windows 生成二进制包的最后一个8.1.X版本。鼓励Windows用户迁移到8.2.X或更高的版本， 因为在8.2.X中有Windows特定的不可移植的修复。8.1.X将继续支持其他平台。

E.114.1. 迁移到版本 8.1.11

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.114.2. 修改列表

- 阻止索引中的函数用用户运行 `VACUUM` 、 `ANALYZE` 等的权限执行 (Tom)

  在索引表达式和部分索引谓词中使用的函数在制作一个新的表条目时评估。 很早我们就知道，如果一个人修改了一个属于不可信用户的表，那么会引起特洛伊木马执行的风险。 （请注意，触发器、缺省、检查约束等也会引起相同类型的风险。） 但是索引中的函数还会引起额外的危险，因为它们将被日常维护操作执行，比如 `VACUUM FULL` ，它通常是在超级用户账户下自动执行的。例如， 一个邪恶的用户可以通过设置一个特洛伊木马索引定义并且等待下一个日常清理， 使用超级用户的权限执行代码。该修复安排标准的维护操作（包括 `VACUUM` 、 `ANALYZE` 、 `REINDEX` 和 `CLUSTER` ） 作为表所有者而不是调用的用户执行，使用和早已为 `SECURITY DEFINER` 函数使用的一样的权限转换机制。为了阻止绕开这个安全机制， `SET SESSION AUTHORIZATION` 和 `SET ROLE` 的执行现在禁止在一个 `SECURITY DEFINER` 内容中。 (CVE-2007-6600)

- 修复正则表达式包中的各种bug (Tom, Will Drewry)

  适当配置的正则表达式模式可能会引起崩溃，无限的或接近无限的循环， 和/或大量的内存消耗，所有这些都造成服务拒绝从不可信的源接受正则搜索模式的应用的危害。 (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- 要求使用 `/contrib/dblink` 的非超级用户只使用密码认证作为一个安全机制 (Joe)

  在8.1.10中出现的该修复是不完整的，因为它只堵住了一些 `dblink` 函数的漏洞。 (CVE-2007-6601, CVE-2007-3278)

- 更新时区数据文件到tzdata版本2007k （特别的，最近的Argentina的改变） (Tom)
- 改善非C环境中规划器对LIKE/正则估计的处理 (Tom)
- 修复一些 `WHERE false AND var IN (SELECT ...)` 的情况下规划器的失败 (Tom)
- 保留通过 `ALTER TABLE ... ALTER COLUMN TYPE` 重建的索引的表空间 (Tom)
- 让归档恢复总是启动一个新的WAL时间线，而不是只在使用恢复停止时间时启动 (Simon)

  这避免了尝试重写最后WAL段的一个现有归档拷贝的极端情况的风险， 并且看起来比原先的定义更加简单和干净。
- 让 `VACUUM` 在表太小而没什么用处时不要使用所有的 `maintenance_work_mem` (Alvaro)
- 修复使用一个多字节数据库编码时 `translate()` 中潜在的崩溃 (Tom)
- 为超过68年的间隔修复 `extract(epoch from interval)` 中的溢出 (Tom)
- 修复PL/Perl，以在UTF-8正则表达式在一个受信任的函数中使用时不失败 (Andrew)
- 修复PL/Perl处理何时平台的Perl定义类型 `bool` 为 `int` ， 而不是 `char` (Tom)

  理论上这会在任何地方发生，非标准的Perl建立是这样的……直到Mac OS X 10.5。
- 修复PL/Python在长的异常消息上不会崩溃 (Alvaro)
- 修复pg_dump以正确的处理拥有与它们的父表不同的缺省表达式的继承子表 (Tom)
- 修复 `PGPASSFILE` 引用一个不是普通文件的文件时 libpq的崩溃 (Martin Pitt)
- ecpg解析器修复 (Michael)
- 让 `contrib/pgcrypto` 防止OpenSSL库在键长于128位时失败； 这至少是在一些Solaris版本上的情况 (Marko Kreen)
- 让 `contrib/tablefunc` 的 `crosstab()` 处理NULL行id为它本身的一个类别，而不是崩溃 (Joe)
- 修复 `tsvector` 和 `tsquery` 输出例程，以正确的逃逸反斜杠 (Teodor, Bruce)
- 修复 `to_tsvector()` 在大的输入字符串上的崩溃 (Teodor)
- 当重新生成 `configure` 脚本时，要求使用Autoconf 的一个特定版本 (Peter)

  这只影响开发者和打包者。该修改是为了阻止意外的使用未测试的 Autoconf和PostgreSQL版本的组合。 如果你真的想要使用一个不同的Autoconf版本， 你可以删除版本校验，但是结果如何就是你自己的责任了。

E.115. 版本 8.1.10

> 发布日期: 2007-09-17

这个版本包含各种自8.1.9以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.115.1. 迁移到版本 8.1.10

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.115.2. 修改列表

- 当一个事务插入行，然后在接近同一个表中的并发 `VACUUM` 的结尾退出时，阻止索引损坏 (Tom)
- 让 `CREATE DOMAIN ... DEFAULT NULL` 正确的工作 (Tom)
- 允许 `interval` 数据类型接受只有毫秒或微妙组成的输入 (Neil)
- 加速rtree索引插入 (Teodor)
- 修复过度的记录SSL错误消息 (Tom)
- 修复日志，这样日志消息在使用系统记录过程时从不交叉 (Andrew)
- 修复 `log_min_error_statement` 记录耗完内存时的崩溃 (Tom)
- 修复不正确的处理一些外键极端情况 (Tom)
- 阻止 `REINDEX` 和 `CLUSTER` 由于尝试处理其他会话的临时表而失败 (Alvaro)
- 更新时区数据库规则，尤其是New Zealand即将到来的修改 (Tom)
- Windows套接字改善 (Magnus)
- 在Windows上的日志时间戳中压缩时区名 (`%Z`)， 因为可能的编码错误匹配 (Tom)
- 要求使用 `/contrib/dblink` 的非超级用户只使用密码认证作为一个安全措施 (Joe)

E.116. 版本 8.1.9

> 发布日期: 2007-04-23

这个版本包含各种自8.1.8以来的修复，包括一个安全修复。 关于8.1主版本的新特性信息，请查看Section E.125。

E.116.1. 迁移到版本 8.1.9

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.116.2. 修改列表

- 在 `search_path` 中支持明确的替换临时表模式， 并且禁止为函数和操作符搜索它 (Tom)

  这需要允许安全定义函数设置一个 `search_path` 的真实安全的值。 如果没有，一个非特权的SQL用户可以用安全定义函数的权限使用临时对象执行代码 (CVE-2007-2138)。参阅 `CREATE FUNCTION` 获取更多信息。

- `/contrib/tsearch2` 崩溃修复 (Teodor)
- 要求 `COMMIT PREPARED` 在和事务预备的同一个数据库中执行(Heikki)
- 在 `VACUUM FULL` 如何处理 `UPDATE` 链中修复潜在的数据损坏bug (Tom, Pavan Deolasee)
- 规划器修复，包括改善外连接和位图扫描选择逻辑 (Tom)
- 修复扩大一个哈希索引期间的PANIC (在8.1.6中引入的bug)
- 修复POSIX风格的时区规格以遵从新的USA DST规则 (Tom)

E.117. 版本 8.1.8

> 发布日期: 2007-02-07

这个版本包含一个来自8.1.7的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.117.1. 迁移到版本 8.1.8

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.117.2. 修改列表

- 在约束和功能性索引中为类型长度删除过分限制的检查 (Tom)

E.118. 版本 8.1.7

> 发布日期: 2007-02-05

这个版本包含各种自8.1.6以来的修复，包括一个安全修复。 关于8.1主版本的新特性信息，请查看Section E.125。

E.118.1. 迁移到版本 8.1.7

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.118.2. 修改列表

- 删除允许连接的用户读取后端内存的安全缺陷 (Tom)

  该缺陷包括抑制一个SQL函数返回它声明的数据类型的正常检查， 和修改一个表字段的数据类型 (CVE-2007-0555, CVE-2007-0556)。 这些错误可以很容易的利用来导致一个后端崩溃， 并且实际上可能被用来读取用户不应该能够访问的数据库内容。

- 修复btree索引页分裂中可能会由于选择一个不可行的分裂点而失败的罕见bug (Heikki Linnakangas)
- 为带有许多表的数据库提高 `VACUUM` 性能 (Tom)
- 修复自动清理以避免在非可连接的数据库中留下非参数事务ID (Alvaro)

  这个bug只影响8.1分支。

- 修复由 `UNION` 触发的罕见的Assert()崩溃 (Tom)
- 为超过三个字节长度的UTF8序列加强多字节字符处理的安全 (Tom)
- 修复由于尝试同步早已删除的文件而在Windows上发生的伪造的"权限被拒绝"失败 (Magnus, Tom)
- 修复更新一个早已在使用的PL/pgSQL函数时可能的崩溃 (Tom)

E.119. 版本 8.1.6

> 发布日期: 2007-01-08

这个版本包含各种自8.1.5以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.119.1. 迁移到版本 8.1.6

运行8.1.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.119.2. 修改列表

- 改善在AIX上 `getaddrinfo()` 的处理 (Tom)

  这修复了在其他事情上开始统计收集器的问题。

- 修复pg_restore，以处理包含带有注释的大对象（blobs）的tar格式备份 (Tom)
- 修复 `VACUUM` 中的"未能重新找到父键"错误 (Tom)
- 在服务器重启期间清理 `pg_internal.init` 缓存文件 (Simon)

  这避免了在PITR恢复之后缓存文件可能包含旧的数据的危险。

- 修复 `VACUUM` 截断一个超过十亿字节边界的大关系的竞态条件 (Tom)
- 修复在行级别锁上引起不必要的死锁错误的bug (Tom)
- 修复影响多个十亿字节哈希索引的bug (Tom)
- 修复Windows信号处理中可能的死锁 (Teodor)
- 修复构造一个由多个空元素组成的 `ARRAY[]` 时的错误 (Tom)
- 修复连接期间ecpg内存泄露 (Michael)
- 修复Darwin (OS X)编译 (Tom)
- 对于新的initdb安装， `to_number()` 和 `to_char(numeric)` 现在是 `STABLE` ， 不是 `IMMUTABLE` (Tom)

  这是因为 `lc_numeric` 可以潜在的改变这些函数的输出。

- 改善索引对使用括号的正则表达式的使用 (Tom)

  这也提高了psql `\d` 的性能。

- 更新时区数据库

  特别的，这影响了澳大利亚和加拿大的夏令时规则。

E.120. 版本 8.1.5

> 发布日期: 2006-10-16

这个版本包含各种自8.1.4以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.120.1. 迁移到版本 8.1.5

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.120.2. 修改列表

- 不允许聚集函数在 `UPDATE` 命令中，除了在子SELECT中 (Tom)

  这样一个聚集的行为是不可预知的，并且在8.1.X中会导致一个崩溃，所以已经禁用了。SQL标准也不允许这样的聚集。

- 修复无类型的字符串看成是ANYARRAY时的内核转储
- 修复执行一个 `COMMIT` 或 `ROLLBACK` 时， 扩展查询协议的时间日志中的内核转储
- 修复查询包含一个返回多行的SQL函数时AFTER触发器的错误处理 (Tom)
- 修复 `ALTER TABLE ... TYPE` ，为 `USING` 子句重新检查 `NOT NULL` (Tom)
- 修复 `string_to_array()` ，以处理分隔符字符串的重复匹配

  例如， `string_to_array('123xx456xxx789', 'xx')` 。

- 为 `AM` / `PM` 格式修复 `to_timestamp()` (Bruce)
- 修复决定是否需要 `ANALYZE` 的自动清理的计算 (Alvaro)
- 为psql的 `\d` 命令修复模式匹配中的极端情况
- 修复/contrib/ltree中的索引损坏错误 (Teodor)
- ecpg中的多个稳健性修复 (Joachim Wieland)
- 修复/contrib/dbmirror中的反斜杠逃逸
- /contrib/dblink和/contrib/tsearch2中的小修复

- 哈希表和位图索引扫描中的效率提升 (Tom)
- 修复Windows上统计收集器的不稳定性 (Tom, Andrew)
- 修复`statement_timeout`，使其在Win32上使用合适的单位 (Bruce)

  在以前的Win32 8.1.X版本中，延迟通过一个100的因子关闭。
- 修复MSVC和Borland C++编译器 (Hiroshi Saito)
- 修复AIX和Intel编译器 (Tom)
- 修复连续归档中少见的bug (Tom)

E.121. 版本 8.1.4

> 发布日期: 2006-05-23

这个版本包含各种自8.1.3以来的修复，包括对极其严重的安全问题的修补。 关于8.1主版本的新特性信息，请查看Section E.125。

E.121.1. 迁移到版本 8.1.4

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

针对CVE-2006-2313和CVE-2006-2314中描述的SQL注入攻击的完全安全可能需要在应用的代码中修改。 如果你有应用嵌入了不可信的字符串到SQL命令中，你应该尽快检测它们， 以确保它们使用推荐的逃逸技术。在大多数情况下，应用应该使用库或驱动 （比如libpq的 `PQescapeStringConn()` ） 提供的子例程执行字符串逃逸，而不是依赖于*ad hoc*代码完成逃逸。

E.121.2. 修改列表

- 修改服务器以在所有情况下都拒绝无效编码的多字节字符 (Tatsuo, Tom)

  虽然PostgreSQL已经朝这个方向发展了一段时间了， 但是检查现在才一致的应用到所有编码和所有文本输入中， 并且现在总是提示错误而不只是警告。这个修改防范了CVE-2006-2313 中描述的SQL注入类型的攻击。

- 拒绝在字符串文本中不安全的使用 `\'`

  作为服务器端防范CVE-2006-2314中描述的SQL注入类型的攻击， 服务器现在只接受 `''` 而不是 `\'` 作为SQL字符串字面值中ASCII单引号的表示。缺省的， 只在 `client_encoding` 设置为仅客户端的编码 （SJIS、BIG5、GBK、GB18030或UHC）时拒绝 `\'` ， 这也是SQL注入有可能发生的情节。一个新的配置参数 `backslash_quote` 可用于在需要时调整这个行为。请注意，针对CVE-2006-2314 的完全安全可能需要客户端侧的修改；但是 `backslash_quote` 的目的是在一定程度上让不安全的客户端明显。

- 修改libpq的字符串逃逸例程，意识到编码注意问题和 `standard_conforming_strings`

  这为CVE-2006-2313和CVE-2006-2314中描述的安全问题修复了使用libpq 的应用，并且也提前防范了规划的到SQL标准字符串文本语法的转换。 同时使用多个PostgreSQL连接的应用应该迁移到 `PQescapeStringConn()` 和 `PQescapeByteaConn()` ， 以确保为每个数据库

连接中使用的设置做了正确的逃逸。 "手动"做字符串逃逸的应用应该修改为依赖于库例程。

- 修复pgcrypto中的弱键选择 (Marko Kreen)

  fortuna PRNG重播逻辑中的错误会导致一个可预知的会话秘钥在某些情况下被 `pgp_sym_encrypt()` 选择。这只影响没有使用OpenSSL的构造。

- 修复一些不正确的编码转换函数

  `win1251_to_iso` 、 `win866_to_iso` 、 `euc_tw_to_big5` 、 `euc_tw_to_mic` 、 `mic_to_euc_tw` 都在变化范围上有损坏。

- 清理字符串中剩下的 `\'` 的使用 (Bruce, Jan)
- 让自动清理在 `pg_stat_activity` 中可见 (Alvaro)
- 禁用 `full_page_writes` (Tom)

  在某些情况下，让 `full_page_writes` 关闭会导致崩溃恢复失败。 一个适当的修复将在8.2中出现；现在只是将它禁用了。

- 各种规划器修复，尤其是位图索引扫描和MIN/MAX最优化 (Tom)
- 修复合并连接中不正确的最优化 (Tom)

  外连接有时会发出多次未匹配行的拷贝。

- 修复在相同的事务中使用和修改plpgsql函数的崩溃
- 修复B-Tree索引被截断情况下的WAL重放
- 为包含 `|` 的模式修复 `SIMILAR TO` (Tom)
- 修复 `SELECT INTO` 和 `CREATE TABLE AS` ， 以在缺省表空间中创建表，而不是在基础目录中 (Kris Jurka)
- 修复服务器，以正确的使用自定义DH SSL参数 (Michael Fuhr)
- 改善快速排序性能 (Dann Corbit)

  目前这个代码只在Solaris上使用了。

- 修复x86系统上的OS/X Bonjour (Ashley Clark)
- 修复各种小的内存泄露
- 修复密码提示在某些Win32系统上的问题 (Robert Kinberg)
- 改善pg_dump对域的缺省值的处理

- 修复pg_dumpall以合理的处理命名相同的用户和组（只在从早于8.1的服务器上转储时是可能的） (Tom)

  用户和组将带有`LOGIN`权限被合并到一个角色中。 以前合并的角色不会有`LOGIN`权限，使其不能用作一个用户。

- 修复pg_restore `-n`，以作为记录工作 (Tom)

E.122. 版本 8.1.3

> 发布日期: 2006-02-14

这个版本包含各种自8.1.2以来的修复，包括一个非常严重的安全问题。 关于8.1主版本的新特性信息，请查看Section E.125。

E.122.1. 迁移到版本 8.1.3

运行8.1.X的用户不需要转储/恢复。不过，如果你是从一个早于8.1.2的版本升级而来， 那么请查看8.1.2的版本声明。

E.122.2. 修改列表

- 修复允许任何登陆的用户为任何其他数据库用户id `SET ROLE` 的错误 (CVE-2006-0553)

  由于不充分的有效性检查，用户会开发特殊的情况， `SET ROLE` 通常用于在错误之后恢复以前的角色设置。例如，这允许普通用户请求超级用户身份。 权限的增加只在8.1.0-8.1.2中存在风险。不过，在所有回退到7.3的版本中， 在 `SET SESSION AUTHORIZATION` 中有一个相关的错误， 允许非特权的用户使服务器崩溃，如果已经启用断言编译了（这不是缺省的）。 感谢Akio Ishida报告这个问题。

- 修复自动插入的行中行可见性逻辑的错误 (Tom)

  在少数情况下，当前命令插入的行可能被看做早已是有效的，而此时不应该是这样的。 修复在8.0.4、7.4.9和7.3.11版本中创建的错误。

- 修复在pg_clog和pg_subtrans文件创建期间会导致"文件早已存在" 错误的竞态条件 (Tom)
- 修复缓存失效消息正好在错误时间到达导致崩溃的情况 (Tom)
- 为预备语句中的 `UNKNOWN` 参数适当的检查 `DOMAIN` 约束 (Neil)
- 确保 `ALTER COLUMN TYPE` 以正确的顺序处理 `FOREIGN KEY` 、 `UNIQUE` 和 `PRIMARY KEY` 约束 (Nakano Yoshihisa)
- 修复以允许恢复拥有交叉模式引用自定义操作符或操作符类的转储 (Tom)
- 允许pg_restore在 `COPY` 失败之后正确的继续； 以前它尝试将剩余的 `COPY` 数据看做SQL命令 (Stephen Frost)
- 当没有指定数据目录时，修复pg_ctl `unregister` 的崩溃 (Magnus)

- 修复libpq `PQprint` HTML标签 (Christoph Zwerschke)
- 修复在AMD64和PPC上的ecpg崩溃 (Neil)
- 允许 `SETOF` 和 `%TYPE` 一起在函数结果类型声明中使用
- 如果错误发生在参数在PL/python中传递期间，那么正确的恢复 (Neil)
- 修复 `plperl_return_next` 中的内存泄露 (Neil)
- 修复PL/perl在Win32的环境上匹配后端的处理 (Andrew)
- 各种优化器修复 (Tom)
- 修复 `log_min_messages` 在Win32上设置为 `DEBUG3` 或 `postgresql.conf` 之外时的崩溃 (Bruce)
- 为Win32、Cygwin, OS X、AIX修复pgxs `-L` 的库路径声明 (Bruce)
- 在检查Win32管理员权限时检查SID是否启用 (Magnus)
- 适当的拒绝超出范围的数据输入 (Kris Jurka)
- 可移植性修复，以在配置期间测试 `finite` 和 `isinf` 的存在 (Tom)
- 通过避免每个数据行的内核调用，提高 `COPY IN` 经过libpq的速度 (Alon Goldshuv)
- 提高 `/contrib/tsearch2` 索引创建的速度 (Tom)

E.123. 版本 8.1.2

> 发布日期: 2006-01-09

这个版本包含各种自8.1.1以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.123.1. 迁移到版本 8.1.2

运行8.1.X的用户不需要转储/恢复。不过， 如果你受到本地环境或下面描述的 plperl问题的影响， 你可能需要在升级之后 `REINDEX` 文本字段上的索引。

E.123.2. 修改列表

- 修复Windows代码，这样如果在ShmemBackendArray中没有更多的空间， 主进程将继续而不是退出 (Magnus)

  如果太多连接请求差不多一起到达，那么以前的代码行为将导致拒绝服务的情况。 这只应用到Windows端口。

- 修复在8.0中引入的bug，该bug会允许读缓冲区返回一个早已用过的页面作为新的页面，潜在的导致丢失最近提交的数据 (Tom)

- 修复在一个事务外部或一个失败的事务内部发出的协议级别的描述信息 (Tom)

- 为认为不同字符组合相等的环境，如Hungarian，修复字符串比较 (Tom)

  这可能需要 `REINDEX` 来修复文本字段上现有的索引。

- 在主进程启动期间设置本地环境变量，以确保plperl 在稍后不会改变本地环境

  这避免了postmaster启动时的环境变量和initdb 说明的不同时发生的问题。在这些条件下，plperl 的任何使用都有可能导致损坏索引。如果你遇到了这些， 你可能需要 `REINDEX` 文本字段上现有的索引。

- 允许安装目录更灵活的重新定位 (Tom)

  以前的版本只在所有安装目录路径都相同除了最后一个组件不同时支持重新定位。

- 阻止由于使用 `ISO-8859-5` 和 `ISO-8859-9` 编码而引起的崩溃 (Tatsuo)

- 修复strpos()和正则表达式中处理某些很少使用的Asian多字节字符集中长期存在的bug (Tatsuo)

- 修复COPY CSV模式认为任何 `\.` 为结束拷贝数据的bug

  新的代码要求 `\.` 在每个文档中都单独的显示在一行中。

- 让COPY CSV模式引用 `\.` 的文本数据值， 以确保它不会被解释为结束数据的标记 (Bruce)

- 各种修复返回 `RECORD` 的函数 (Tom)

- 修复 `postgresql.conf` 的处理， 这样一个没有换行符的最后一行能够正确的处理 (Tom)

- 修复 `/contrib/pgcrypto` gen_salt中的bug， 它导致不能为MD5和XDES算法使用所有可用的盐空间 (Marko Kreen, Solar Designer)

  Blowfish和标准的DES的盐没有受到影响。

- 修复处理表达式索引时自动清理的崩溃

- 修复 `/contrib/dblink`， 当指定的字段数量和查询实际返回的数量不同时， 抛出一个错误而不是崩溃 (Joe)

E.124. 版本 8.1.1

> 发布日期: 2005-12-12

这个版本包含各种自8.1.0以来的修复。关于8.1主版本的新特性信息， 请查看Section E.125。

E.124.1. 迁移到版本 8.1.1

运行8.1.X的用户不需要转储/恢复。

E.124.2. 修改列表

- 修复外部链接条件不正确的最优化 (Tom)
- 修复在包含通过优化器优化了的子查询的情况下，错误报告字段名的问题 (Tom)
- 修复在包括CHECK约束、toast字段和索引的情节中的更新失败 (Tom)
- 修复从错误中恢复之后的后台写问题 (Tom)

  发现后台写在写入错误之后泄露缓存针。虽然这在它本身不致命， 但是这可能会导致稍后VACUUM命令难以理解的堵塞。
- 如果客户端在当前事务早已终止时发送绑定的协议消息，阻止失败。
- `/contrib/tsearch2` 和 `/contrib/ltree` 修复 (Teodor)
- 修复语言中翻译的错误消息需要重新排序单词的问题，比如Turkish； 还有输出字符串意外截断和最小可能的bigint值错误显示的问题 (Andrew, Tom)

  这个问题只在使用了我们的 `port/snprintf.c` 代码的平台上出现， 如果给出了 `--enable-nls` 则该代码包括了BSD变种，可能还有其他。 另外，一个不同形式的翻译错误消息问题可能会在Windows上出现， 取决于使用了 `libintl` 的哪个版本。
- 为 `to_char(time)` 和 `to_char(interval)` 重新允许 `AM` / `PM` 、 `HH` 、 `HH12` 和 `D` 格式说明符。 (`to_char(interval)` 应该可能使用 `HH24` 。) (Bruce)
- AIX、HPUX和MSVC编译修复 (Tom, Hiroshi Saito)
- 优化器改善 (Tom)
- 在Windows NO_SYSTEM_RESOURCES错误之后重新尝试文件读取和写入 (Qingqing Zhou)

- 阻止ANALYZE表达式索引期间autovacuum的崩溃 (Alvaro)
- 修复ON COMMIT DELETE ROWS临时表的问题
- 修复触发器修改SELECT DISTINCT查询的输出时的问题
- 添加了8.1.0版本声明，说明了如何迁移无效的 `UTF-8` 字节序列 (Paul Lindner)

E.125. 版本 8.1

> 发布日期: 2005-11-08

E.125.1. 概要

这个版本中的主要修改：

改善到共享缓冲区缓存的并发访问 (Tom)

访问共享缓冲区缓存被认定为是一个重要的可扩展性问题，尤其是在多个CPU的系统上。 在这个版本中，锁定的方式是以缓冲区管理器已经被检查以减少锁征用和提高可扩展性来完成的。 缓冲区管理器也已经被修改为使用一个"时钟扫描"替换机制。

允许索引扫描使用一个中间内存位图 (Tom)

在以前的版本中，只有单个索引可以用来在表上查找。有了这个特性，如果一个查询有 `WHERE tab.col1 = 4 and tab.col2 = 9`，并且在 `col1` 和 `col2` 上没有多字段索引，但是在 `col1` 上有一个索引， 并且在 `col2` 上有另一个索引，那么有可能搜索两个索引并在内存中组合结果， 然后为同时匹配 `col1` 和 `col2` 限制条件的行执行堆栈获取。 这在有许多非结构化的查询，可能创建匹配所有访问条件的索引的环境中是非常有用的。 即使是单个索引，位图扫描也是有用的，因为他们减少了所需要的随机访问的数量；位图索引扫描对于检索完整表的相当大的分数是高效的，而普通索引扫描则不行。

添加了两阶段提交 (Heikki Linnakangas, Alvaro, Tom)

两阶段提交允许事务在几个计算机上"准备"，并且一旦所有计算机都成功的准备了它们的事务（没有失败），那么所有事务都可以提交。即使一个机器在准备之后崩溃了， 准备的事务也可以在该机器重启之后提交。新的语法包括 `PREPARE TRANSACTION` 和 `COMMIT/ROLLBACK PREPARED` 。还添加了一个新的系统视图 `pg_prepared_xacts` 。

创建一个新的替换用户和组的角色系统

角色是用户和组的一个组合。像用户，它们可以有登陆功能，像组， 一个角色可以拥有其他角色作为成员。角色基本上删除了用户和组之间的不同。 例如，一个角色可以：

- 有登陆功能 (可选)
- 拥有对象
- 持有数据库对象的访问权限
- 从它作为成员的其他角色上继承权限

一旦用户登陆到一个角色，她获得了登陆角色加上任何继承的角色的能力， 并且可以使用 `SET ROLE` 来切换到其他角色（她在其中是一个成员）。 这个特性是SQL标准角色的概念的一个概括。这个修改还用新的角色能力目录 `pg_authid` 和 `pg_auth_members` 替换了 `pg_shadow` 和 `pg_group` 。 老的表被重新定义为新角色表上的只读视图。

为 `MIN()` 和 `MAX()` 自动使用索引 (Tom)

在以前的版本中，为 `MIN()` 或 `MAX()` 使用索引的唯一方式是重写查询 `SELECT col FROM tab ORDER BY col LIMIT 1` 。 索引使用现在是自动发生的。

移动 `/contrib/pg_autovacuum` 到主服务器 (Alvaro)

整合自动清理到服务器，允许它在数据库服务器的同步中自动启动和停止， 并允许自动清理在 `postgresql.conf` 中配置。

使用 `SELECT ... FOR SHARE` 添加共享的行级别锁 (Alvaro)

然而PostgreSQL的MVCC锁允许 `SELECT` 永不被写入锁定，并且因此不需要为典型的操作共享行锁， 共享的锁对于请求共享的行锁的应用是有用的。 特别是它减少了由于参照完整性检查增加的锁请求。

在共享的对象上添加依赖性，尤其是角色 (Alvaro)

依赖性机制的扩展阻止了角色仍然拥有数据库对象时被删除。 以前有可能意外的"孤儿"对象被删除它们的所有者。 虽然这可以被恢复，但是它是杂乱的并且会使人不愉快。

为分区表提升性能 (Simon)

新的 `constraint_exclusion` 配置参数避免了在约束表示没有匹配行存在的子表上查找。

这允许表分区的基本类型。如果子表存储独立的键范围，并且强制使用适当的 `CHECK` 约束，那么优化器将在约束保证在子表中没有匹配的行存在时跳过子表访问。

E.125.2. 迁移到版本 8.1

对于那些想要从任何以前的版本中迁移数据的用户来说， 使用pg_dump的一个转储/恢复是必需的。

8.0版本宣布间隔的 `to_char()` 函数将在8.1中删除。不过， 因为没有更好的API被推荐， `to_char(interval)` 已经在8.1中被加强， 并且将保留在服务器中。

观察下列的不兼容性：

- `add_missing_from` 现在缺省是假 (Neil)

  缺省的，如果表用在一个没有 `FROM` 引用的查询中，我们现在产生一个错误。 老的行为仍然可用，但是参数必须设置为'true'。

为了加载一个现有的转储文件，如果转储包含任何使用隐式 `FROM` 语法创建的视图或规则，那么将 `add_missing_from` 设置为true是必须的。 这应该是一个一次性的烦恼，因为PostgreSQL 8.1 将转换这样的视图和规则到标准的显式 `FROM` 语法。 随后的转储将因此不会有这个问题。

- 让 `float4` / `float8` / `oid` 零长度字符串(`''`)的输入抛出一个错误， 而不是将它看做零 (Neil)

  这个修改与当前整数的零长度字符串的处理是一致的。 这个修改计划在8.0中宣布。

- `default_with_oids` 现在缺省为假 (Neil)

  这个选项设置为假，用户创建的表不再有一个OID字段，除非在 `CREATE TABLE` 中指定了 `WITH OIDS` 。尽管OID已经存在于PostgreSQL 的所有版本中，但是因为它们只有四字节长度并且计数器是基于所有安装的数据库共享的， 所以它们的使用是受限制的。唯一标识行的首选方式是通过序列和 `SERIAL` 类型， 这个自从PostgreSQL 6.4开始就已经支持了。

- 添加 `E''` 语法，这样最后普通字符串可以正确的对待反斜杠 (Bruce)

  目前PostgreSQL作为引入一个特殊的逃逸序列来处理字符串文本中的反斜杠， 比如 `\n` 或 `\010` 。虽然这允许特殊值的简单入口， 但是它是不标准的并且使得从其他数据库中移植应用更加困难。因为这个原因， PostgreSQL工程计划删除反斜杠在字符串中的特殊含义。 为了向后兼容和想要特殊处理反斜杠的用户，创建了一个新的字符串语法。 这个新的字符串语法的格式是在开始字符串的单引号之前写一个 `E` ， 例如 `E'hi\n'` 。虽然这个版本没有改变反斜杠在字符串中的处理， 但是它确实添加了新的配置参数来帮助用户迁移应用到未来的版本：

  - `standard_conforming_strings` —这个版本正确的对待反斜杠为普通字符串？
  - `escape_string_warning` —在普通的字符串（非E）中警告反斜杠

  `standard_conforming_strings` 值是只读的。应用可以检索该值来知道反斜杠是如何处理的。 （参数的存在也可以作为一个支持 `E''` 字符串语法的指示。）在未来的版本中， `standard_conforming_strings` 将为真，意味着反斜杠在非E字符串中将按照字面值对待。为了准备这个修改，在需要特殊处理反斜杠的地方使用 `E''` 字符串， 并且打开 `escape_string_warning` 来找到需要使用 `E''` 转换的额外的字符串。另外，使用两个单引号(`''`)在一个字符串中嵌入一个字面的单引号， 而不是PostgreSQL支持的反斜杠单引号(`\'`)语法。 前者是符合标准的，并且不需要使用 `E''` 字符串语法。你也可以使用 `$$` 字符串语法，它不特殊对待反斜杠。

- 让 `REINDEX DATABASE` 重新索引数据库中的所有索引 (Tom)

  以前， `REINDEX DATABASE` 只重建系统表的索引。 这个新的行为看起来更加直观。一个新的命令 `REINDEX SYSTEM` 提供只重建系统表的老的功能。

- 只读大对象描述符现在服从MVCC快照语义

  当一个大对象是用 `INV_READ` （不是 `INV_WRITE` ）打开的时， 来自描述符的数据读取现在将反应调用 `lo_open()` 的查询使用了事务快照时大对象的状态的"快照"。 要获取总是返回最后提交的数据的老的行为，在 `lo_open()` 的模式标志中包含 `INV_WRITE` 。

- 为序列函数的参数添加适当的依赖 (Tom)

  在以前的版本中，传递到 `nextval()` 、 `currval()` 和 `setval()` 的序列名是作为简单文本字符串存储的， 意味着重命名或删除一个在 `DEFAULT` 子句中使用的序列会使子句无效。 这个版本将所有新建的序列函数参数作为内部OID存储，允许它们追踪序列重命名， 并且添加阻止不正确的序列删除的依赖关系信息。这也使得 `DEFAULT` 子句免疫模式重命名和搜索路径改变。

  一些应用可能依赖于运行时查找序列名的老的行为。这通过明确的转换参数为 `text` 仍然可以实现，例如 `nextval('myseq'::text)` 。

  8.1之前的数据库转储加载到8.1将使用老的基于文本的表示， 并且因此没有OID存储参数的特性。不过，更新包含基于文本的 `DEFAULT` 子句的数据库是可能的。首先，保存这个查询到一个文件中，比如 `fixseq.sql` ：

  ```
  SELECT  'ALTER TABLE ' ||
     pg_catalog.quote_ident(n.nspname) || '.' ||
     pg_catalog.quote_ident(c.relname) ||
     ' ALTER COLUMN ' || pg_catalog.quote_ident(a.attname) ||
     ' SET DEFAULT ' ||
     regexp_replace(d.adsrc,
                    $$val\(\(('[^']*')::text\)::regclass$$,
                    $$val(\1$$,
                    'g') ||
     ';'
  FROM    pg_namespace n, pg_class c, pg_attribute a, pg_attrdef d
  WHERE   n.oid = c.relnamespace AND
     c.oid = a.attrelid AND
     a.attrelid = d.adrelid AND
     a.attnum = d.adnum AND
     d.adsrc ~ $$val\(\('[^']*'::text\)::regclass$$;
  ```

  然后，在一个数据库中运行该查询，找到需要哪个调整，例如对于数据库 `db1` ：

  ```
  psql -t -f fixseq.sql db1
  ```

  这将显示 `ALTER TABLE` 命令需要转换数据库到新的基于OID的表示。 如果命令看起来合理，运行这个来更新该数据库：

  ```
  psql -t -f fixseq.sql db1 | psql -e db1
  ```

  这个过程必须在每个要更新的数据库中重复进行。

- 在psql中，将未加引号的 `\{digit}+` 序列看做是八进制的 (Bruce)

在以前的版本中， `\{digit}+` 序列被看做是小数点，并且只有 `\0{digit}+` 被看做是八进制的。这个修改是为了一致性。

- 为前缀和后缀 `%` 和 `^` 操作符删除语法生产

  这些从未记录和复杂的负数的模数操作符(`%`)的使用。

- 让多边形的 `&<` 和 `&>` 与盒子的"over"操作符一致 (Tom)

- `CREATE LANGUAGE` 可以忽略提供的参数，为了支持来自 `pg_pltemplate` 的信息

  定义了一个新的系统目录 `pg_pltemplate` ， 携带关于过程语言首选定义的信息（比如它们是否有验证器函数）。 当一个条目存在于创建语言的目录中时， `CREATE LANGUAGE` 将忽略所有它的参数除了语言名并使用目录信息。 采取这个措施是因为废弃的语言定义被老的转储文件加载而增加问题。 截止到8.1，pg*dump*将只是作为 *`CREATE LANGUAGE`* `_name` 转储过程语言定义，依赖于加载时模板条目的存在。 我们期待这将成为一个更加不会过时的表示。

- 让 `pg_cancel_backend(int)` 返回一个 `boolean` 而不是一个 `integer` (Neil)

- 一些用户在加载UTF-8数据到8.1.X时遇到问题。这是因为以前的版本允许无效的UTF-8字节序列输入到数据库中，而这个版本只接受有效的UTF-8序列。 纠正转储文件的一个方式是运行命令 `iconv -c -f UTF-8 -t UTF-8 -o cleanfile.sql dumpfile.sql` 。 `-c` 选项删除无效的字符序列。 两个文件的差异将显示无效的序列。 `iconv` 读取整个输入文件到内存中， 这样它可能为了处理需要使用split将转储分成多个较小的文件。

E.125.3. 额外的修改

下面你将发现PostgreSQL 8.1和以前的主版本间详细的额外的修改。

E.125.3.1. 性能改善

- 提高GiST和R-tree索引性能(Neil)

- 改善优化器，包括自动调整哈希连接的大小 (Tom)

- 彻底检查几个方面的内部API

- 修改WAL记录CRC从64位到32位 (Tom)

  我们觉得计算64位CRC的额外开销非常大，并且获得的可靠性并不足以为它辩解。

- 阻止在WAL页面中写入大的空间隙 (Tom)

- 改善SMP机器上的自旋锁行为，尤其是Opterons (Tom)

- 允许非连续的索引字段在多字段索引上使用 (Tom)

例如，这允许一个在字段a、b、c上的索引在一个带有 `WHERE a = 4 and c = 10` 的查询中使用。

- 为 `CREATE TABLE AS` / `SELECT INTO` 跳过WAL日志 (Simon)

  因为 `CREATE TABLE AS` 期间的崩溃会导致表在恢复期间被删除， 没有理由WAL记录该表被加载了。（不过，如果启用了WAL归档，记录仍然会发生。）

- 允许并发的GiST索引访问 (Teodor, Oleg)

- 添加配置参数 `full_page_writes` 控制写入全部页面到WAL (Bruce)

  为了阻止部分磁盘写入损坏数据库，PostgreSQL 写了每个数据库磁盘页面的完整拷贝来WAL它在一个检查点之后被修改的第一个时间。 这个选项为了更快的速度关闭了该功能。这对于使用电池后备的磁盘缓存来说是安全的， 这种情况下部分页面写入不会发生。

- 当为 `wal_sync_method` 使用 `O_SYNC` 时， 如果可用则使用 `O_DIRECT` (Itagaki Takahiro)

  `O_DIRECT` 导致磁盘写入绕过内核缓存，对于WAL写入来说，这提高了性能。

- 提高 `COPY FROM` 性能 (Alon Goldshuv)

  这是通过在更大的语块中读取 `COPY` 输入完成的，而不是挨个读取字符。

- 提高了 `COUNT()` 、 `SUM` 、 `AVG()` 、 `STDDEV()` 和 `VARIANCE()` 的性能 (Neil, Tom)

E.125.3.2. 服务器的变化

- 阻止由于事务ID (XID)环绕式处理引起的问题 (Tom)

  当事务计数器达到环绕的点时，服务器现在将发出警告。如果计数器即将达到环绕的点时， 服务器将停止接受查询。这保证了数据在需要的清理执行之前不会丢失。

- 修复OID计数器已经环绕式处理之后，对象IDs (OIDs)与现有系统对象冲突的问题 (Tom)

- 添加 `VACUUM` 期间需要增加 `max_fsm_relations` 和 `max_fsm_pages` 的警告 (Ron Mayer)

- 添加 `temp_buffers` 配置参数， 允许用户为临时表访问确定本地缓存区域的大小 (Tom)

- 添加会话启动时间和客户端IP地址到 `pg_stat_activity` (Magnus)

- 为位图扫描调整 `pg_stat` 视图 (Tom)

  一些字段的含义发生了微妙的变化。

- 加强了 `pg_locks` 视图 (Tom)

- 客户端侧 `PREPARE` 和 `EXECUTE` 的日志查询 (Simon)

- 允许Kerberos名字和用户名在 `postgresql.conf` 的规定中大小写敏感 (Magnus)
- 添加配置参数 `krb_server_hostname` ， 这样服务器主机名可以作为服务主体的一部分指定 (Todd Kover)

  如果没有设置，任何服务主体匹配keytab中的任意条目都有可能被使用。 这是这个版本中的新的Kerberos匹配行为。
- 为毫秒时间戳(`%m`)和远程主机(`%h`)添加 `log_line_prefix` 选项 (Ed L.)
- 为GiST索引添加WAL日志 (Teodor, Oleg)

  GiST索引现在对于崩溃和时间点恢复来说是安全的。
- 当我们执行 `pg_stop_backup()` 时， 删除老的 `*.backup` 文件 (Bruce)

  这阻止了大量的 `*.backup` 文件存在于 `pg_xlog/` 中。
- 添加配置参数为闲置、间隔和计数控制TCP/IP保持活动的时间 (Oliver Jowett)

  这些值可以修改来允许对丢失的客户端连接更快速的检测。
- 添加每用户和每数据库连接限制 (Petr Jelinek)

  使用 `ALTER USER` 和 `ALTER DATABASE` ， 现在可以强制限制作为一个特殊用户或作为一个特殊数据库并发连接的最大会话数量。 设置限制为0禁用用户或数据库连接。
- 在64位的机器上允许超过两千兆字节的共享内存和每后端的工作内存 (Koichi Suzuki)
- 新增系统目录 `pg_pltemplate` ，允许在转储文件中重写废弃的过程语言定义 (Tom)

E.125.3.3. 查询修改

- 添加临时视图 (Koju Iijima, Neil)
- 修复 `HAVING` 没有任何聚集函数或 `GROUP BY` ， 这样查询返回一个组 (Tom)

  以前，这样的情况会将 `HAVING` 子句看做和 `WHERE` 子句相同。 这是不规范的。
- 添加 `USING` 子句允许额外的表指定为 `DELETE` (Euler Taveira de Oliveira, Neil)

  在以前的版本中，没有明确的方法指定用于 `DELETE` 语句中的连接的额外的表。 对于这个目的， `UPDATE` 早已有了一个 `FROM` 子句。
- 在后端和ecpg字符串中添加对 `\x` 十六进制逃逸的支持 (Bruce)

  这正像标准C `\x` 逃逸语法。八进制逃逸早就支持了。
- 添加 `BETWEEN SYMMETRIC` 查询语法 (Pavel Stehule)

这个特征允许`BETWEEN`比较不请求第一个值小于第二个值。 例如， `2 BETWEEN [ASYMMETRIC] 3 AND 1`返回假， 而`2 BETWEEN SYMMETRIC 3 AND 1`返回真。`BETWEEN ASYMMETRIC`早就支持了。

- 添加`NOWAIT`选项到`SELECT ... FOR UPDATE/SHARE` (Hans-Juergen Schoenig)

  当`statement_timeout`配置参数允许一个查询接受超过一定数量的时间被取消，`NOWAIT`选项允许查询在`SELECT ... FOR UPDATE/SHARE` 命令不能立即获得一个行锁时尽快被取消。

E.125.3.4. 对象操作的改变

- 追踪共享对象的依赖性 (Alvaro)

  PostgreSQL允许全局表（用户、数据库、表空间） 引用多个数据库中的信息。这为全局表额外的添加了依赖关系信息，这样，例如， 用户所有权可以跨数据库追踪，一个在任意数据库中拥有某些东西的用户不再被删除。 依赖关系追踪早已为数据库本地对象存在了。

- 允许受限制的`ALTER OWNER`命令被对象所有者执行 (Stephen Frost)

  以前的版本只允许超级用户修改对象所有者。现在，如果执行该命令的用户拥有该对象， 并且可以作为新的所有者创建它，那么所有权就可以转移（也就是说， 该用户是新的拥有角色的一员，并且该角色拥有重新创建该对象所需要的CREATE权限）。

- 为某些对象类型（表、函数、类型）添加`ALTER`对象`SET SCHEMA`能力 (Bernd Helmle)

  这允许对象迁移到不同的模式中。

- 添加`ALTER TABLE ENABLE/DISABLE TRIGGER`禁用触发器 (Satoshi Nagayasu)

E.125.3.5. 工具命令的变化

- 允许`TRUNCATE`在一个命令中截断多个表 (Alvaro)

  由于参照完整性检查，不允许截断是参照完整性约束的一部分的表。 使用这个新功能， `TRUNCATE`可以用来截断这个样的表， 如果两个表都包含在一个参照完整性约束中，那么会在一个`TRUNCATE`命令中都被截断。

- 在`COPY CSV`模式中正确的处理回车和换行 (Andrew)

  在版本8.0中， `CSV COPY TO`中的回车和换行是以一种不一致的方式处理的。 （这在TODO列表中记录了。）

- 添加`COPY WITH CSV HEADER`，允许标题作为`COPY` 中的第一行 (Andrew)

这允许处理公共 `CSV` 在数据文件的第一行放置字段名的用法。 对于 `COPY TO` ， 第一行包含该字段名，对于 `COPY FROM` ， 忽略第一行。

- 在Windows上，在 `EXPLAIN ANALYZE` 中显示更好的次秒级精度 (Magnus)
- 添加触发时间显示到 `EXPLAIN ANALYZE`

  以前的版本包含触发器的执行时间作为总的执行时间的一部分， 但是没有单独显示它。现在有可能看到在每个触发器中花费了多少时间。

- 在 `COPY` 中添加对 `\x` 十六进制逃逸的支持 (Sergey Ten)

  以前的版本只支持八进制逃逸。

- 让 `SHOW ALL` 包括变量描述 (Matthias Schmidt)

  `SHOW` 变量名仍然只显示变量的值，不包括变量描述。

- 让initdb创建一个新的称为 `postgres` 的标准数据库， 并转变工具为标准查询使用 `postgres` 而不是 `template1` (Dave)

  在以前的版本中， `template1` 作为工具（像createuser） 的缺省连接使用，也作为新数据库的模板。这导致 `CREATE DATABASE` 有时会失败，因为如果任何人连接了模板数据库，那么新的数据库就不能创建。 有了这个改变，缺省连接数据库现在是 `postgres` ， 意味着在 `CREATE DATABASE` 期间，极少可能有人正在使用 `template1` 。

- 通过移动 `/contrib/reindexdb` 到服务器， 创建新的reindexdb命令行工具 (Euler Taveira de Oliveira)

E.125.3.6. 数据类型和函数的变化

- 为数组类型添加 `MAX()` 和 `MIN()` 聚集 (Koju Iijima)
- 修复 `CC` 和 `YY` 字段都被使用了时， `to_date()` 和 `to_timestamp()` 合理的行为 (Karel Zak)

  如果格式声明包含 `CC` 并且年的声明是 `YYY` 或更长， 那么忽略 `CC` 。如果年的声明是 `YY` 或更短， 那么解释 `CC` 为上个世纪。

- 添加 `md5(bytea)` (Abhijit Menon-Sen)

  `md5(text)` 早已存在了。

- 添加对基于 `power(numeric, numeric)` 的 `numeric ^ numeric` 的支持

  该函数早就存在了，但是没有分配操作符给他。

- 通过正确的在计算时截断商，修复 `NUMERIC` 的模数 (Bruce)

在以前的版本中，大值的模数有时会因为商的圆整返回负的结果。

- 添加了一个函数 `lastval()` (Dennis Björklund)

  `lastval()` 是 `curval()` 的一个简化版本。 它根据当前会话执行的最后一个 `nextval()` 或 `setval()` 调用，自动的确定正确的序列名。

- 添加了 `to_timestamp(DOUBLE PRECISION)` (Michael Glaesemann)

  转换自1970年以来的Unix秒到一个 `TIMESTAMP WITH TIMEZONE` 。

- 添加了 `pg_postmaster_start_time()` 函数 (Euler Taveira de Oliveira, Matthias Schmidt)

- 允许在 `AT TIME ZONE` 中充分使用时区名字， 不只是以前可用的短列表 (Magnus)

  以前， `AT TIME ZONE` 只支持时区名的一个预定义的列表。 现在任何支持的时区名都可以使用，例如：

  ```
  SELECT CURRENT_TIMESTAMP AT TIME ZONE 'Europe/London';
  ```

  在上面的查询中，使用的时区基于实际上影响提供的日期的夏令时规则调整。

- 添加 `GREATEST()` 和 `LEAST()` variadic函数 (Pavel Stehule)

  这些函数接受可变数量的参数并返回这些参数的最大或最小值。

- 添加了 `pg_column_size()` (Mark Kirkwood)

  这返回一个字段的存储尺寸，可能是被压缩的。

- 添加了 `regexp_replace()` (Atsushi Ogawa)

  这允许正则表达式替换，像sed。一个可选的标识参数允许全局选择（替换所有） 和大小写敏感模式。

- 修复间隔除法和乘法 (Bruce)

  以前的版本有时返回不正当的结果，像 `'4 months'::interval / 5` 返回 `'1 mon -6 days'` 。

- 修复时间戳、时间和间隔输出中的舍入行为 (Tom)

  这修复了秒字段显示为 `60` 而不是增长高阶字段的一些情况。

- 添加一个单独的天字段到类型 `interval` ， 这样一天的间隔可以不同于24小时的间隔 (Michael Glaesemann)

包含夏令时调整的天不是24小时长，通常是23或25小时。这个修改创建了一个 "这么多天"的间隔和"这么多小时"的间隔之间概念上的不同。 添加 `1 day` 到一个时间戳现在给出和下一天相同的本地时间， 即使两天之间有一个夏令时调整，而添加 `24 hours` 将给出一个不同的本地时间。例如，在US DST规则下：

```
'2005-04-03 00:00:00-05' + '1 day' = '2005-04-04 00:00:00-04'
'2005-04-03 00:00:00-05' + '24 hours' = '2005-04-04 01:00:00-04'
```

- 添加了 `justify_days()` 和 `justify_hours()` (Michael Glaesemann)

  这些函数，分别的，调整天到一个合适数量的全月和日， 调整小时到一个合适数量的全天和小时。

- 移动 `/contrib/dbsize` 到后端，并重命名一些函数 (Dave Page, Andreas Pflug)

  - `pg_tablespace_size()`
  - `pg_database_size()`
  - `pg_relation_size()`
  - `pg_total_relation_size()`
  - `pg_size_pretty()`

  `pg_total_relation_size()` 包括索引和TOAST表。

- 为访问集群目录的只读文件添加函数 (Dave Page, Andreas Pflug)

  - `pg_stat_file()`
  - `pg_read_file()`
  - `pg_ls_dir()`

- 添加 `pg_reload_conf()` 强制配置文件的重载 (Dave Page, Andreas Pflug)
- 添加 `pg_rotate_logfile()` 强制服务器日志文件的循环 (Dave Page, Andreas Pflug)
- 修改 `pg_stat_*` 视图包含TOAST表 (Tom)

E.125.3.7. 编码和环境的变化

- 重命名一些编码使其更加一致和遵守国际标准 (Bruce)

  - `UNICODE` 现在是 `UTF8`
  - `ALT` 现在是 `WIN866`
  - `WIN` 现在是 `WIN1251`

    - `TCVN` 现在是 `WIN1258`

  原先的名字仍然工作。

- 添加对 `WIN1252` 编码的支持 (Roland Volkmann)
- 添加对四字节 `UTF8` 字符的支持 (John Hansen)

  以前只支持一、二、三字节 `UTF8` 字符。这对于支持一些汉字字符设置尤其重要。

- 允许 `EUC_JP` 和 `SJIS` 之间的直接转换，以提高性能 (Atsushi Ogawa)
- 允许UTF8编码在Windows上工作 (Magnus)

  这是通过映射UTF8到Windows本地的UTF16实现做到的。

E.125.3.8. 一般服务器端语言的变化

- 修复 `ALTER LANGUAGE RENAME` (Sergey Yatskevich)
- 允许函数特征，像严格和活泼，通过 `ALTER FUNCTION` 来修改 (Neil)
- 增加函数参数的最大数量到100 (Tom)
- 允许SQL和PL/pgSQL函数使用 `OUT` 和 `INOUT` 参数 (Tom)

  `OUT` 是函数返回值的一个可替换的方式。取代使用 `RETURN`， 可以通过分配参数声明为 `OUT` 或 `INOUT` 来返回值。 这在一些情况下通常更简单，尤其是需要返回多个值时。 从一个函数中返回多个值在以前的版本中是可能的，这大大的简化了该过程。 （该特性在未来的版本中将扩展为其他服务器端语言。）

- 移动语言处理器函数到 `pg_catalog` 模式

  如果要求，这使得它更容易删除公共模式。

- 添加 `SPI_getnspname()` 到SPI (Neil)

E.125.3.9. PL/pgSQL服务器端语言的变化

- 彻底检查PL/pgSQL函数的内存管理 (Neil)

  每个函数的分析树现在存储在一个单独的内存空间中。 当不再需要它时，这允许这个内存很容易的回收利用。

- 在 `CREATE FUNCTION` 时检查函数语法，而不是在运行时 (Neil)

  以前，大多数语法错误只在该函数被执行时报告。

- 允许 `OPEN` 打开非 `SELECT` 查询，像 `EXPLAIN` 和 `SHOW` (Tom)

- 不再要求函数发出 `RETURN` 语句 (Tom)

  这是新添加的 `OUT` 和 `INOUT` 功能的副产品。 当不需要提供函数的返回值时，可以省略 `RETURN` 。

- 添加对可选 `INTO` 子句到PL/pgSQL的 `EXECUTE` 语句的支持 (Pavel Stehule, Neil)
- 让 `CREATE TABLE AS` 设置 `ROW_COUNT` (Tom)
- 定义 `SQLSTATE` 和 `SQLERRM` 返回当前异常的 `SQLSTATE` 和错误消息 (Pavel Stehule, Neil)

  这些变量只在异常块中定义。

- 允许到 `RAISE` 语句的参数为表达式 (Pavel Stehule, Neil)
- 添加一个循环的 `CONTINUE` 语句 (Pavel Stehule, Neil)
- 允许块和循环标签 (Pavel Stehule)

E.125.3.10. PL/Perl服务器端语言的变化

- 允许大的结果集有效的返回 (Abhijit Menon-Sen)

  这允许函数使用 `return_next()` 来避免在内存中建立整个结果集。

- 允许一次一行检索查询结果 (Abhijit Menon-Sen)

  这允许函数使用 `spi_query()` 和 `spi_fetchrow()` 避免在内存中累加整个结果集。

- 如果服务器编码是 `UTF8` ，那么强制PL/Perl将字符串作为 `UTF8` 处理 (David Kamholz)
- 为PL/Perl添加一个验证器函数 (Andrew)

  这允许语法错误在定义时被报告，而不是在执行时报告。

- 当函数返回一个数组类型时，允许PL/Perl返回一个Perl数组 (Andrew)

  这主要是映射PostgreSQL数组到Perl数组。

- 允许Perl非致命的警告产生 `NOTICE` 消息 (Andrew)
- 允许启用Perl的 `strict` 模式 (Andrew)

E.125.3.11. psql的变化

- 添加 `\set ON_ERROR_ROLLBACK` 以允许事务中的语句发生错误而不影响剩余的事务 (Greg Sabino Mullane)

  这基本上是通过包裹每个语句在一个子事务中实现的。

- 在psql变量中添加对 `\x` 十六进制字符串的支持 (Bruce)

  八进制逃逸早就支持了。

- 添加对 `troff -ms` 输出格式的支持 (Roger Leigh)
- 允许历史文件位置通过 `HISTFILE` 控制 (Andreas Seltenreich)

  这允许存储每个数据库历史的配置。

- 阻止 `\x` （扩展模式）影响 `\d tablename` 的输出 (Neil)
- 添加 `-L` 选项到psql以记录会话 (Lorne Sunley)

  添加这个选项是因为一些操作系统没有简单的命令行活动记录功能。

- 让 `\d` 显示索引的表空间 (Qingqing Zhou)
- 允许psql帮助(`\h`) 基于正确的帮助信息做一个更好的猜测 (Greg Sabino Mullane)

  这允许用户仅仅添加 `\h` 到语法错误查询的前面，获得支持的语法的帮助。 以前任何超出命令名的额外的查询文本都必须使用 `\h` 删除。

- 添加 `\pset numericlocale` 以允许数字在识别环境的格式中输出 (Eugen Nedelcu)

  例如，使用 `C` 环境 `100000` 将被输出为 `100,000.0` ， 而欧洲环境可能输出这个值为 `100.000,0` 。

- 当服务器版本号和psql的版本号不同时，让启动标语显示两者 (Bruce)

  还有，如果服务器和psql来自不同的主版本，那么将会显示一个警告。

E.125.3.12. pg_dump的变化

- 添加 `-n` / `--schema` 开关到pg_restore (Richard van den Berg)

  这允许只恢复指定模式中的对象。

- 允许pg_dump转储大对象，即使是在文本模式中 (Tom)

  有了这个变化，大对象现在总是被转储了；前者 `-b` 开关是一个空操作。

- 允许pg_dump转储大对象的一个一致的快照 (Tom)
- 为大对象转储的评论 (Tom)
- 添加 `--encoding` 到pg_dump (Magnus Hagander)

  这允许数据库以一个不同于服务器编码的编码转储。 这在传递转储到一个有不同编码的机器时是有价值的。

- 依赖 `pg_pltemplate` 过程语言 (Tom)

  如果过程语言的调用处理器在 `pg_catalog` 模式中，那么pg*dump* 不转储该处理器。相反的，它使用 `CREATE LANGUAGE` `_*name* `转储该语言，依赖于` pg_pltemplate`目录提供语言的创建参数和加载时间。

E.125.3.13. libpq的变化

- 添加一个 `PGPASSFILE` 环境变量指定口令文件的文件名 (Andrew)
- 添加 `lo_create()`， 类似于 `lo_creat()`， 但是允许指定大对象的OID (Tom)
- 让libpq在 `malloc()` 失败时， 一致的返回一个错误到客户端应用 (Neil)

E.125.3.14. 源代码的变化

- 修复pgxs，以支持重定位安装的建立
- 为使用Intel编译器的Itanium处理器添加自旋锁支持 (Vikram Kalsi)
- 为Windows添加Kerberos 5支持 (Magnus)
- 添加了Chinese FAQ (laser@pgsqldb.com)
- 重命名Rendezvous为Bonjour以匹配OS/X特性重命名 (Bruce)
- 在Darwin上添加对 `fsync_writethrough` 的支持 (Chris Campbell)
- 流线化信息在服务器、优化器和锁系统中的路径 (Tom)
- 允许pg_config使用MSVC编译 (Andrew)

  这在使用MSVC编译DBD::Pg时是需要的。

- 删除对Kerberos V4的支持 (Magnus)

  Kerberos 4有安全缺陷并且不再维护了。

- 代码清理（EnterpriseDB执行Coverity静态分析）
- 修改 `postgresql.conf` 使用文档缺省 `on` / `off`， 而不是 `true` / `false` (Bruce)
- 增强pg_config，能够报告更多的构建时的值 (Tom)
- 允许libpq在Windows上的建立是线程安全的 (Dave Page)
- 允许IPv6连接在Windows上使用 (Andrew)
- 添加关于I/O子系统可靠性的服务器管理文档 (Bruce)

- 从 `gist.h` 中移动私有的声明到 `gist_private.h` (Neil)

  在以前的版本中， `gist.h` 包含公共的GiST API （用来被GiST索引实现的作者使用）和一些被GiST本身的实现使用的私有声明。 后者已经被移动到一个独立的文件， `gist_private.h` 。 大多数GiST索引实现应该是不受影响的。

- 彻底检查GiST内存管理 (Neil)

  GiST方法现在总是在一个短期存活的内存空间中调用。因此，通过 `palloc()` 的内存分配将被自动回收利用，所以GiST索引实现不需要通过 `pfree()` 手动释放分配的内存。

E.125.3.15. 贡献版的变化

- 添加了 `/contrib/pg_buffercache` 贡献版模板 (Mark Kirkwood)

  这显示了缓冲区缓存的内容，为了调试和性能调优的目的。

- 删除了 `/contrib/array` ，因为它是废弃的 (Tom)
- 清理 `/contrib/lo` 模块 (Tom)
- 移动 `/contrib/findoidjoins` 到 `/src/tools` (Tom)
- 从 `/contrib/cube` 中删除了 `<<` 、 `>>` 、 `&<` 和 `&>` 操作符

  这些操作没什么用处。

- 改善 `/contrib/btree_gist` (Janko Richter)
- 改善 `/contrib/pgbench` (Tomoaki Sato, Tatsuo)

  现在有一个设施测试用户给出的SQL命令脚本，而不是只有一个硬线连接的命令序列。

- 改善 `/contrib/pgcrypto` (Marko Kreen)
  - OpenPGP对称秘钥和公共秘钥加密的实现

    RSA和Elgamal公共秘钥算法都支持。

  - 独立构建：包括SHA256/384/512 hashes, Fortuna PRNG
  - OpenSSL构建:支持3DES，使用内部的AES带有OpenSSL < 0.9.7
  - 从 `configure` 的结果中获取构建参数(OpenSSL, zlib)

    不再需要编辑 `Makefile` 。

  - 删除对 `libmhash` 和 `libmcrypt` 的支持

E.126. 版本 8.0.26

> 发布日期: 2010-10-04

这个版本包含各种自8.0.25以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

这预计是8.0.X系列的最后一个PostgreSQL版本。 建议用户尽快更新到新版本。

E.126.1. 迁移到版本 8.0.26

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.22的版本升级而来， 那么请参阅8.0.22的版本声明。

E.126.2. 修改列表

- 为每个在PL/Perl和PL/Tcl中调用的SQL userid使用一个单独的解释器 (Tom Lane)

  这个修改阻止由损坏另一个SQL用户身份在相同的会话中稍后执行的Perl或Tcl代码引起的安全问题 （例如，在一个 `SECURITY DEFINER` 函数中）。大多数脚本语言提供多种方式， 比如重定义目标函数调用的标准函数或操作符。如果没有这个修改， 任何拥有Perl或Tcl语言使用权限的SQL 用户基本上可以用目标函数的所有者的SQL权限做任何事情。

  这个修改的代价是Perl和Tcl函数之间的沟通变得更加困难了。 为了提供一个逃逸出口，PL/PerlU和PL/TclU函数继续使用每个会话只有一个解释器。 不认为这是一个安全问题，因为所有这样函数的执行早已在数据库超级用户的受信任级别。

  有可能自称提供受信任的执行的第三方过程语言也有相似的安全问题。 我们建议为了安全鉴定，联系你依赖的任何PL的作者。

  感谢Tim Bunce指出这个问题 (CVE-2010-3433)。

- 阻止 `pg_get_expr()` 中可能的崩溃，通过不允许它被非系统目录字段的参数调用 (Heikki Linnakangas, Tom Lane)

- 修复"不能处理未计划的子查询"错误 (Tom Lane)

  当子查询包含一个别名引用连接，该引用连接扩展到一个包含另一个子查询的表达式时会发生这个问题。

- 当并非所有返回的行都是相同的行类型时，防止函数返回一组记录 (Tom Lane)

- 当写fsync的锁文件内容时，要小心（ `postmaster.pid` 也是套接字锁文件） (Tom Lane)

如果机器在主进程启动后不久就崩溃，那么这个疏忽会导致损坏锁文件内容。 随后会阻止主进程启动成功，直到手动删除锁文件。

- 当设定XID为深度嵌套子事务时，避免递归 (Andres Freund, Robert Haas)

  如果这里限制堆栈空间，那么原来的代码会导致一个崩溃。

- 修复 `log_line_prefix` 的 `%i` 逃逸， 这可能在后台启动时就产生垃圾 (Tom Lane)
- 修复启用归档时， `ALTER TABLE ... SET TABLESPACE` 中可能的数据损坏 (Jeff Davis)
- 允许 `CREATE DATABASE` 和 `ALTER DATABASE ... SET TABLESPACE` 被查询取消中断 (Guillaume Lelarge)
- 在PL/Python中，防止从 `PyCObject_AsVoidPtr` 和 `PyCObject_FromVoidPtr` 中产生空指针 (Peter Eisentraut)
- 改善 `contrib/dblink` 对包含删除了的字段的表的处理 (Tom Lane)
- 修复 `contrib/dblink` 中出现"重复的连接名称" 错误之后的连接泄露 (Itagaki Takahiro)
- 修复 `contrib/dblink` ，正确的处理连接名字长于62字节的情况 (Itagaki Takahiro)
- 更新编译基础结构和文档，以反映源代码仓库从CVS搬到了Git (Magnus Hagander and others)
- 更新时区数据文件到tzdata版本2010l，因为DST规律在Egypt和Palestine改变了； 也更正了Finland的历史。

  这个改变也为两个Micronesian时区添加了新的名字： Pacific/Chuuk现在优先于Pacific/Truk（优先的缩写是CHUT不是TRUT）， Pacific/Pohnpei优先于Pacific/Ponape。

E.127. 版本 8.0.25

> 发布日期: 2010-05-17

这个版本包含各种自8.0.24以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

PostgreSQL社区将在2010年7月停止发放8.0.X版本系列的更新。 建议用户尽快更新到一个新的版本。

E.127.1. 迁移到版本 8.0.25

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.22的版本升级而来， 那么请参阅8.0.22的版本声明。

E.127.2. 修改列表

- 使用一个开放标记应用到整个解释器，替代使用 `Safe.pm` ， 强制 `plperl` 中的限制 (Tim Bunce, Andrew Dunstan)

  最近的发展向我们证实：依赖 `Safe.pm` 来标记 `plperl` 可以信赖太不安全了。这个改变整个删除了 `Safe.pm` 的使用， 支持使用一个总是应用开放代码标记的单独的解释器。 这个改变令人愉快的副作用包括：现在在 `plperl` 中以普通方式使用Perl的 `strict` 编程是可能的了，并且Perl的 `$a` 和 `$b` 变量在短例程中像预期的那样工作，并且函数编译明显的更快了。 (CVE-2010-1169)

- 阻止PL/Tcl执行 `pltcl_modules` 中不受信任的代码 (Tom)

  PL/Tcl自动从数据库表中加载Tcl代码的特性可能会被特洛伊代码攻击利用， 因为没有谁可以创建或插入那个表的限制。这个修改禁用了该特性， 除非 `pltcl_modules` 属于超级用户。 （不过，没有对表上的权限进行检查， 所以实际需要一个较小安全模块表的安装仍然可以获得合适的权限来信任非超级用户。） 另外，阻止加载代码到不受限制的"普通"Tcl解释器中， 除非我们实在是想要执行一个 `pltclu` 函数。 (CVE-2010-1170)

- 不允许非特权用户重置超级用户参数设置 (Alvaro)

  以前，如果一个非特权用户为自己运行 `ALTER USER ... RESET ALL` ， 或者为他拥有的数据库运行 `ALTER DATABASE ... RESET ALL` ， 会为该用户或该数据库删除所有特殊参数设置，即使其中有只支持超级用户可改的参数。 现在， `ALTER` 将只删除该用户有权限更改的参数。

- 如果后端关闭发生在向日志条目添加 `CONTEXT` 时，避免后端关闭期间可能的崩溃 (Tom)

  在某些情况下，内容输出函数会失败，因为要输出一条日志信息时，当前事务早已回滚。

- 更新pl/perl的 `ppport.h` 为现代Perl版本 (Andrew)
- 修复pl/python中的各种内存泄露 (Andreas Freund, Tom)
- 当展开一个引用自身的变量时，阻止psql中的无限递归 (Tom)
- 确保 `contrib/pgstattuple` 函数迅速的取消中断 (Tatsuhito Kasahara)
- 让服务器启动适当的处理 `shmget()` 为一个现有的共享内存段返回 `EINVAL` 的情况 (Tom)

  这个行为在BSD驱动的内核包含OS X上观察到。它导致一个完全错误误导的启动失败，抱怨共享内存请求尺寸太大。

- 更新时区数据文件到tzdata版本2010j，因为DST规律在Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia改变了；也为Taiwan做了历史修正。

E.128. 版本 8.0.24

> 发布日期: 2010-03-15

这个版本包含各种自8.0.23以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

PostgreSQL社区将在2010年7月停止更新8.0.X版本系列。 鼓励用户尽快更新到新的版本。

E.128.1. 迁移到版本 8.0.24

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.22的版本升级而来， 那么请参阅8.0.22的版本声明。

E.128.2. 修改列表

- 添加新的配置参数 `ssl_renegotiation_limit` ， 控制我们多久做一次SSL连接的会话秘钥协商 (Magnus)

  可以设置为0来完全禁止协商，如果使用了一个破碎的SSL库可能需要这样做。 特别的，一些供应商为导致协商尝试失败的CVE-2009-3555提供了紧急补丁。

- 修复当尝试从子事务启动失败中恢复时可能的崩溃 (Tom)

- 修复使用保存点和客户端编码与服务器编码不同时相关的服务器内存泄露 (Tom)

- 让 `substring()` 的 `bit` 类型对待任意负的长度为 "所有剩余的字符串" (Tom)

  以前的代码只以这种方式对待-1，并且会为其他负值产生一个无效的结果值， 可能会导致崩溃 (CVE-2010-0442)。

- 修复当输出位宽度比给定的整数（不是8位的倍数）宽时， integer-to-bit-string转换正确的处理第一部分的字节 (Tom)

- 修复一些正则表达式匹配慢的情况 (Tom)

- 修复备份历史文件中的 `STOP WAL LOCATION` 条目， 当结束位置正好在段边界时报告下一条WAL段的名字 (Itagaki Takahiro)

- 当读取 `pg_hba.conf` 和相关的文件时，如果 `@` 出现在双引号中， 那么不要将 `@something` 当做文件包含请求；另外， 永不将 `@` 本身当做文件包含请求 (Tom)

这阻止了一个角色或者数据库名字以 `@` 开头时的奇怪的行为。 如果你需要包含一个路径名包含空格的文件，你也可以这样做， 但是必须写 `@"/path to/file"` ，而不是让双引号包围整个构造。

- 如果一个路径以 `pg_hba.conf` 和相关文件中的包含目标命名， 那么阻止某些平台上的无限循环 (Tom)
- 修复复合字段设置为NULL时的plpgsql失败 (Tom)
- 在PL/Python中添加 `volatile` 标记，以避免可能的编译器特定错误行为 (Zdenek Kotala)
- 确保PL/Tcl完全初始化Tcl解释器 (Tom)

  这个监督的唯一已知症状是Tcl `clock` 命令在使用Tcl 8.5或更新时会错误行为。

- 当指定太多的关键字段到 `dblink_build_sql_*` 函数时， 阻止 `contrib/dblink` 中的崩溃 (Rushabh Lathia, Joe Conway)
- 修复粗心的内存管理引起的 `contrib/xml2` 中的各种崩溃 (Tom)
- 更新时区文件到tzdata版本2010e，因为DST规律在 Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa发生了变化。

E.129. 版本 8.0.23

> 发布日期: 2009-12-14

这个版本包含各种自8.0.22以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.129.1. 迁移到版本 8.0.23

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.22的版本升级而来， 那么请参阅8.0.22的版本声明。

E.129.2. 修改列表

- 防止由索引函数改变会话本地状态引起的间接安全风险 (Gurjeet Singh, Tom)

  这个修改阻止早已不变的索引函数可能破坏超级用户的会话 (CVE-2009-4136)。

- 拒绝SSL证书在公用名（CN）字段包含一个嵌入的空字节 (Magnus)

  这阻止了在SSL验证期间无意识的匹配证书到服务器或客户端名 (CVE-2009-4034)。

- 修复后端启动时缓存初始化期间可能的崩溃 (Tom)

- 阻止信号在不安全的时间打断 `VACUUM` (Alvaro)

  这个修改阻止了 `VACUUM FULL` 在它已经提交了它的元组移动之后取消时的恐慌， 还有简单 `VACUUM` 在已经截断表之后中断时的瞬态错误。

- 修复哈希表尺寸估算中由于整数溢出引起的崩溃 (Tom)

  这可能在哈希连接的结果有非常大的规划器估算时发生。

- 修复 `inet` / `cidr` 比较中非常罕见的崩溃 (Chris Mikkelson)

- 修复用于在一个子事务中可以访问的游标的临时文件过早删除 (Heikki)

- 修复PAM口令处理，使其更加健壮 (Tom)

  都知道以前的代码未能组合Linux `pam_krb5` PAM模块和Microsoft Active Directory 作为域控制器。可能在其他的地方也有问题， 因为它对于将传递哪个PAM堆栈参数做了不公平的假设。

- 修复PL/Python中异常处理时罕见的崩溃 (Peter)

- 确保psql的flex模块是用正确的系统头定义编译的 (Tom)

  这修复了 `--enable-largefile` 导致在生成的代码中有不兼容的更改的平台上编译失败的问题。

- 让主进程忽略任何连接请求包中的`application_name`参数， 以提升与未来libpq版本的兼容性 (Tom)

- 更新时区数据文件到tzdata版本2009s， 因为DST规律已经在Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria改变了； 也修正了Hong Kong的历史时间。

E.130. 版本 8.0.22

> 发布日期: 2009-09-09

这个版本包含各种自8.0.21以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.130.1. 迁移到版本 8.0.22

运行8.0.X的用户不需要转储/恢复。不过，如果你在 `interval` 字段上有任何哈希索引， 那么必须在升级到8.0.22之后 `REINDEX` 它们。 另外，如果你是从一个早于8.0.6的版本升级而来，那么请参阅8.0.6的版本声明。

E.130.2. 修改列表

- 不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` 在安全定义函数内部 (Tom, Heikki)

  这包含了一个在前一个补丁中漏掉的情况，不允许 `SET ROLE` 和 `SET SESSION AUTHORIZATION` 在安全定义函数里面。 (参阅CVE-2007-6600)

- 修复出现在外部级别聚集函数参数里的子SELECT的处理 (Tom)

- 为数据类型 `interval` 修复哈希计算 (Tom)

  这纠正了哈希连接在间隔值上的错误结果。也改变了哈希索引在间隔字段上的内容。 如果你有任何这样的索引，你必须在升级之后 `REINDEX` 它们。

- 将 `to_char(..., 'TH')` 当做一个带有 `'HH'` / `'HH12'` 的大写字母顺序后缀 (Heikki)

  以前是作为 `'th'` (小写)处理的。

- 修复 `_x_` 大于2百万并且使用整数日期时间时， `INTERVAL '```_x_` ms'的溢出 (Alex Hunsaker)

- 修复点和线段之间距离的计算 (Tom)

  这导致几个几何运算符不正确的结果。

- 修复 `money` 数据类型以在货币数量没有小数点的本地环境下工作， 比如日本 (Itagaki Takahiro)

- 适当的圆整像 `00:12:57.9999999999999999999999999999` 这样的日期时间输入 (Tom)

- 修复GiST R-tree操作符类中页分裂点的选择很少的问题 (Teodor)

- 修复plperl初始化中的可移植性问题 (Andrew Dunstan)
- 修复如果 `postgresql.conf` 为空，pg_ctl 不要进入无限循环 (Jeff Davis)
- 修复 `contrib/xml2` 的 `xslt_process()`， 适当的处理参数的最大数量 (twenty) (Tom)
- 提高libpq代码的鲁棒性，以从 `COPY FROM STDIN` 期间的错误中恢复 (Tom)
- 避免包括冲突的readline和editline头文件，当两个库都安装了时 (Zdenek Kotala)
- 更新时区数据文件到tzdata版本2009l， 因为DST规律在Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan（只是历史纠正）, Mauritius, Morocco, Palestine, Syria, Tunisia改变了。

E.131. 版本 8.0.21

> 发布日期: 2009-03-16

这个版本包含各种自8.0.20以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.131.1. 迁移到版本 8.0.21

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.131.2. 修改列表

- 当编码转换失败时，阻止错误的递归崩溃 (Tom)

  这个修改扩展了在上两个小版本中相关失败场景的修复。 以前的修复是对于原始报告的问题严密定做的，但是现在我们意识到： 任何由编码转换函数抛出的错误都可能导致在尝试报告错误时的无限递归。 因此如果我们发现我们已经进入一个递归的错误报告情景中， 解决方法是禁用翻译和编码转换并以纯ASCII格式报告任何错误消息。 (CVE-2009-0922)

- 不允许 `CREATE CONVERSION` 给指定的转换函数指定错误的编码 (Heikki)

  这阻止了一个编码转换失败的可能的情形。以前的修改是防御相同情况中其他类型的失败。

- 修复 `to_char()` 给定的格式代码不适合数据参数的类型时的内核转储 (Tom)

- 添加 `MUST` (Mauritius Island Summer Time)到已知时区缩写的缺省列表 (Xavier Bugaud)

E.132. 版本 8.0.20

> 发布日期: 2009-02-02

这个版本包含各种自8.0.19以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.132.1. 迁移到版本 8.0.20

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.132.2. 修改列表

- 提高 `headline()` 函数中URL的处理 (Teodor)
- 改善 `headline()` 函数中超长标题的处理 (Teodor)
- 如果编码转换是用错误的转换函数为指定的编码对创建的， 那么阻止可能的断言失败或错误转换 (Tom, Heikki)
- 避免 `VACUUM` 中小表的不必要的锁定 (Heikki)
- 修复 `contrib/tsearch2` 的 `get_covers()` 函数中未初始化的变量 (Teodor)
- 让所有的文档适当的引用 `pgsql-bugs` 和/或 `pgsql-hackers` ， 代替现在停止使用的 `pgsql-ports` 和 `pgsql-patches` 邮件列表 (Tom)
- 更新时区数据文件到tzdata版本2009a （因为Kathmandu和历史的DST纠正在Switzerland, Cuba）

E.133. 版本 8.0.19

> 发布日期: 2008-11-03

这个版本包含各种自8.0.18以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.133.1. 迁移到版本 8.0.19

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.133.2. 修改列表

- 修复客户端编码不能表示本地化的错误消息时的后端崩溃 (Tom)

  我们以前也记录过相似的问题，但是如果"character has no equivalent" 消息本身不能被转换的话仍然会失败。该修复在检测到这样的情形时禁用本地化， 并发送纯ASCII的错误消息。

- 修复深层嵌套函数从一个触发器调用时可能的崩溃 (Tom)
- 确保一个新定义的PL/pgSQL触发器函数被作为一个普通函数调用时报告一个错误 (Tom)
- 修复单个查询条目匹配文本的第一个单词时产生不正确的tsearch2标题 (Sushant Sinha)
- 修复在 `--enable-integer-datetimes` 编译中使用一个非ISO的日期风格时， 间隔值中不合适的分数秒的显示 (Ron Mayer)
- 确保传递的元组和元组描述符有不同的字段数量时， `SPI_getvalue` 和 `SPI_getbinval` 正确的行为 (Tom)

  当一个表有字段添加或删除时，这种情况是正常的，但是这两个函数没有正确的处理它。 唯一可能的结果是不正确的错误指示。

- 修复ecpg解析 `CREATE USER` (Michael)
- 修复 `pg_ctl restart` 最近的损坏 (Tom)
- 更新时区数据文件到tzdata版本2008i （因为DST规律在Argentina, Brazil, Mauritius, Syria改变了）

E.134. 版本 8.0.18

> 发布日期: 2008-09-22

这个版本包含各种自8.0.17以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.134.1. 迁移到版本 8.0.18

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.134.2. 修改列表

- 本地锁计数器从32扩大到64位 (Tom)

  这是对于在足够长的事务中计数器会溢出，导致意外的 "lock is already held"错误的报告的回应。

- 在执行器启动中添加检查，确保 `INSERT` 或 `UPDATE` 产生的元组将匹配目标表的当前行类型 (Tom)

  `ALTER COLUMN TYPE`， 跟着以前缓存的规划的重新使用， 会产生这种情况。检查阻止了数据损坏和/或接着发生的崩溃。

- 修复日期时间输入函数，以在64位平台上运行时正确的检测整数溢出 (Tom)

- 改善向系统日志写入非常长的日志消息时的性能 (Tom)

- 修复 `SELECT DISTINCT ON` 查询上后向扫描一个游标中的错误 (Tom)

- 修复规划器估算 `GROUP BY` 表达式，不管表达式的内容， 总是在两个组中生成布尔结果 (Tom)

  这比正规 `GROUP BY` 估算某些布尔测试，像 `_col_` `IS NULL`， 显然更加准确。

- 修复PL/Tcl，使其与Tcl 8.5正确的行为，并且更加小心关于发送到或来自Tcl的数据的编码 (Tom)

- 修复PL/Python，使其与Python 2.5一起工作

  这是在8.2开发周期中做的修复的后端接口。

- 改善在未能发送一个SQL命令之后的pg_dump和 pg_restore的错误报告 (Tom)

- 修复pg_ctl以在 `restart` 时适当的保存主进程命令行参数 (Bruce)
- 更新时区数据文件到tzdata版本2008f （因为DST规律在Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine,和Paraguay改变了）

E.135. 版本 8.0.17

> 发布日期: 2008-06-12

这个版本包含一系列对8.0.16的错误修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.135.1. 迁移到版本 8.0.17

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.135.2. 修改列表

- 让`pg_get_ruledef()`给负的常数加上括号 (Tom)

  在这个修复之前，视图或规则中的一个负的常量可能被转储为，假设`-42::integer`， 这样是不正确的：它应该为`(-42)::integer`，因为操作符的优先级规则。 通常这样无关紧要，但是它与另外一个最近的补丁相互影响，导致PostgreSQL 拒绝一个曾经有效的`SELECT DISTINCT`视图查询。 因为这会导致pg_dump输出未能重载，因此它被视为高优先级的修复。 转储输出实际上不正确的发布版本是8.3.1和8.2.7。

E.136. 版本 8.0.16

> 发布日期: never released

这个版本包含各种自8.0.15以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.136.1. 迁移到版本 8.0.16

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.136.2. 修改列表

- 修复 `ALTER TABLE ADD COLUMN ... PRIMARY KEY`， 以便新的字段被正确的检查，看看是否它被初始化为所有非空 (Brendan Jurd)

  以前的版本忽略了检查这个必要条件。

- 修复从多个来自相同祖先的父关系中继承"相同"约束时， 可能的 `CREATE TABLE` 失败 (Tom)

- 修复ISO-8859-5和其他编码之间的转换，以处理Cyrillic "Yo"字符 (`e` 和 `E` 带有两个点） (Sergey Burladyan)

- 修复一些允许未使用的字节出现在它们的结果中包含未初始化的、 不可预计的值的日期类型输入函数 (Tom)

  这会导致两个看起来相同的文字值不被认为相等的失败， 导致分析器抱怨不匹配的 `ORDER BY` 和 `DISTINCT` 表达式。

- 修复正则表达式子串匹配中的极端情况 (`substring(``_string_` from `_pattern_`)) (Tom)

  这个问题出现在：有一个到整个模式的匹配，但是用户指定了一个在括号中的子表达式， 并且该子表达式没有获得匹配的情况。一个例子是： `substring('foo' from 'foo(bar)?')` 。 这应该返回NULL，因为 `(bar)` 没有匹配，但是它错误的返回了整个模式匹配 （也就是 `foo` ） 。

- 更新时区数据文件到tzdata版本2008c （因为DST规律在Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, Argentina/San_Luis, 和Chile改变了）

- 修复ecpg的 `PGTYPEStimestamp_sub()` 函数中的不正确的结果 (Michael)

- 修复输入查询返回一个NULL值时， `contrib/xml2` 的 `xpath_table()` 函数中的内核转储 (Tom)
- 修复 `contrib/xml2` 的makefile，以不覆盖 `CFLAGS` (Tom)
- 修复 `DatumGetBool` macro，使其不在使用gcc 4.3时失败 (Tom)

  这个问题影响返回布尔值的"old style" (V0) C函数。 该修复在8.3中就已经有了，但是返回打补丁的需要在当时并没有意识到。

- 修复长期存在的 `LISTEN` / `NOTIFY` 竞态条件 (Tom)

  在少数情况下，一个刚刚执行了 `LISTEN` 的会话可能不会得到一个通知， 即使预计是有的，因为并发的事务执行 `NOTIFY` 是在提交后观察。

  该修复的副作用是一个执行了至今未提交的 `LISTEN` 命令的事务， 将不会看到该 `LISTEN` 的任何 `pg_listener` 中的行， 而它应该选择查看的；以前的它是查看的。这个行为从未记录过， 但是有可能一些应用依赖于老的行为。

- 修复查询使用哈希索引期间发生错误时的少见的崩溃 (Heikki)
- 修复二月29在公元前的年中的日期时间值的输入 (Tom)
- 中文在此。。。
- 中文在此。。。

  中文在此。。。

- 中文在此。。。

  中文在此。。。

- 中文在此。。。

  中文在此。。。

- 中文在此。。。

  中文在此。。。

E.137. 版本 8.0.15

> 发布日期: 2008-01-07

这个版本包含各种自8.0.14以来的修复，包括对重大安全问题的修复。 关于8.0主版本的新特性信息，请参阅Section E.152。

中文在此。。。

E.137.1. 迁移到版本 8.0.15

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.137.2. 修改列表

- 阻止索引中的函数用用户的权限执行 `VACUUM`，`ANALYZE` 等 (Tom)

  在索引表达式中使用的函数和局部索引谓词是在制作一个新的表项时评估的。 一直以来认为，如果一个人修改了属于不受信任用户的表， 会带来特洛伊木马代码执行的风险。（请注意：触发器、缺省、检查约束等， 构成相同类型的风险。）但是索引中的函数带来额外的风险， 因为它们将被日常维护操作如 `VACUUM FULL` 执行， 而日常维护操作通常是在超级用户账户下自动执行的。例如， 一个不法用户可以通过设置特洛伊木马索引定义，然后等待下一个日常清理， 以超级用户权限执行代码。该修复为标准的维护操作（包括 `VACUUM`、 `ANALYZE`、 `REINDEX` 和 `CLUSTER`） 安排了作为表的所有者而不是调用用户执行，相同的权限切换机制早已用于 `SECURITY DEFINER` 函数。为了阻止绕过这个安全措施， 现在禁止在 `SECURITY DEFINER` 环境中执行 `SET SESSION AUTHORIZATION` 和 `SET ROLE` 。 (CVE-2007-6600)

- 修复了正则表达式包中的各种bug (Tom, Will Drewry)

  适当配置的正则表达式模式可能会引起崩溃，无限或者接近无限的循环， 和/或巨大的内存消耗， 所有这些造成服务器拒绝接受来自不可靠的源的正则表达式搜索模式的危害应用 (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- 需要使用 `/contrib/dblink` 的非超级用户只使用口令认证，作为一个安全措施 (Joe)

  在8.0.14中出现的这个修复是不完整的，因为它只堵住了一些 `dblink` 函数的漏洞。 (CVE-2007-6601, CVE-2007-3278)

- 更新时区数据文件到tzdata版本2007k （尤其是，最近的Argentina修改） (Tom)

- 修复一些`WHERE false AND var IN (SELECT ...)`的规划器失败 (Tom)
- 保留`ALTER TABLE ... ALTER COLUMN TYPE`重建的索引的表空间 (Tom)
- 让归档恢复总是启动一个新的WAL时间轴，而不是只在使用恢复停止时间时使用新的时间轴 (Simon)

  这避免了尝试重写一个现有的最后一个WAL段的归档拷贝的极端情况的风险， 并且看起来比原先的定义更简单、更干净。

- 当表太小以至于没什么用时，让`VACUUM`不使用`maintenance_work_mem` (Alvaro)
- 修复使用多字节数据库编码时， `translate()`潜在的崩溃 (Tom)
- 修复平台的Perl定义类型`bool`作为`int`而不是 `char`时的PL/Perl处理 (Tom)

  虽然理论上会发生在任何地方，但是没有Perl的标准编译是这样做的...直到Mac OS X 10.5。

- 修复PL/Python，使其在长的异常消息上不会崩溃 (Alvaro)
- 修复pg_dump以正确的处理继承的子表和它们的父表有不同的缺省表达式的情况 (Tom)
- 修复了ecpg分析器 (Michael)
- 让`contrib/tablefunc`的`crosstab()` 作为一个类处理空行本身，而不是崩溃 (Joe)
- 修复`tsvector`和`tsquery`输出例程以正确的逃逸反斜杠 (Teodor, Bruce)
- 修复`to_tsvector()`在巨大的输入字符串上的崩溃 (Teodor)
- 当重新生成`configure`脚本时，请求一个特定的Autoconf版本 (Peter)

  这只影响开发者和包装者。这个修改是为了阻止意外的使用未测试的 Autoconf和PostgreSQL版本的组合。 如果你真的想要使用一个不同的Autoconf版本， 你可以删除版本检查，但是结果如何就是你自己的责任了。

E.138. 版本 8.0.14

> 发布日期: 2007-09-17

这个版本包含各种自8.0.13以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.138.1. 迁移到版本 8.0.14

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.138.2. 修改列表

- 阻止事务插入行然后在接近相同表中并发的 `VACUUM` 的结尾退出时的索引损坏 (Tom)
- 让 `CREATE DOMAIN ... DEFAULT NULL` 正确的工作 (Tom)
- 修复过多的记录SSL错误消息 (Tom)
- 修复日志，以便使用系统日志进程时日志消息从不交叉 (Andrew)
- 修复 `log_min_error_statement` 日志用光内存时的崩溃 (Tom)
- 修复一些外键在极端情况下不正确的处理 (Tom)
- 阻止 `CLUSTER` 由于尝试处理其他会话的临时表而失败 (Alvaro)
- 更新时区数据库规则，尤其是New Zealand即将到来的改变 (Tom)
- Windows套接字改善 (Magnus)
- 抑制Windows上日志时间戳中的时区名(`%Z`)， 因为可能的编码错误匹配 (Tom)
- 要求使用 `/contrib/dblink` 的非超级用户只使用口令认证，作为一个安全措施 (Joe)

E.139. 版本 8.0.13

> 发布日期: 2007-04-23

这个版本包含各种自8.0.12以来的修复，包括一个安全修复。关于8.0主版本的新特性信息，请参阅Section E.152。

E.139.1. 迁移到版本 8.0.13

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来，那么请参阅8.0.6的版本声明。

E.139.2. 修改列表

- 支持在 `search_path` 中明确的布置临时表模式，并且禁止函数和操作符搜索它 (Tom)

  这需要允许安全定义函数设置一个 `search_path` 的真正安全的值。没有它，一个没有特权的SQL用户就可以使用临时对象以安全定义函数的权限执行代码 (CVE-2007-2138)。参阅 `CREATE FUNCTION` 获取更多信息。

- `/contrib/tsearch2` 崩溃修复 (Teodor)
- 修复 `VACUUM FULL` 处理 `UPDATE` 链时潜在的数据损坏错误 (Tom, Pavan Deolasee)
- 修复扩大哈希索引期间的恐慌 (在8.0.10中引入的bug)
- 修复POSIX风格规格以遵循新的USA DST规则 (Tom)

E.140. 版本 8.0.12

> 发布日期: 2007-02-07

这个版本包含各种自8.0.11以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.140.1. 迁移到版本 8.0.12

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.140.2. 修改列表

- 删除约束和函数索引中过分严格的类型长度检查 (Tom)

E.141. 版本 8.0.11

> 发布日期: 2007-02-05

这个版本包含各种自8.0.10以来的修复，包括一个安全修复。关于8.0主版本的新特性信息，请参阅Section E.152。

E.141.1. 迁移到版本 8.0.11

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.141.2. 修改列表

- 删除允许连接的用户读取后端内存的安全漏洞 (Tom)

  漏洞包括抑制SQL函数返回它声明的数据类型的正常检查，和改变表字段的数据类型(CVE-2007-0555, CVE-2007-0556)。这些错误可以很容易的被利用， 导致后端崩溃，并且原则上可能被用来读取该用户不能够访问的数据库内容。

- 修复由于选择一个不可行的分裂点导致btree索引页分裂可能失败的少见的错误 (Heikki Linnakangas)
- 修复由 `UNION` 触发的少见的Assert()崩溃 (Tom)
- 加强超过三个字节长度的UTF8序列的多字节字符处理的安全 (Tom)

E.142. 版本 8.0.10

> 发布日期: 2007-01-08

这个版本包含各种自8.0.9以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.142.1. 迁移到版本 8.0.10

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.142.2. 修改列表

- 改善AIX上 `getaddrinfo()` 的处理 (Tom)

  这修复了在其他事情上启动统计收集器的问题。

- 修复 `VACUUM` 中"未能重新找到父键"的错误 (Tom)
- 修复通过 `VACUUM` 在十亿字节边界截断一个大的关系的竞态条件 (Tom)
- 修复影响多个十亿字节哈希索引的错误 (Tom)
- 修复Windows信号处理中可能的死锁 (Teodor)
- 修复构造一个由多个空元素组成的 `ARRAY[]` 时的错误 (Tom)
- 修复连接期间ecpg的内存泄露 (Michael)
- `to_number()` 和 `to_char(numeric)` 现在对于新的initdb安装来说是 `STABLE` ， 不是 `IMMUTABLE` (Tom)

  这是因为 `lc_numeric` 可以潜在的改变这些函数的输出。

- 提升使用括号的正则表达式的索引使用 (Tom)

  这也提升了psql `\d` 的性能。

- 更新时区数据库

  这尤其影响到了Australian和Canadian的夏令时。

E.143. 版本 8.0.9

> 发布日期: 2006-10-16

这个版本包含各种自8.0.8以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.143.1. 迁移到版本 8.0.9

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.143.2. 修改列表

- 修复在规则的WHERE表达式中引用 `NEW` 行值时的崩溃 (Tom)
- 修复无类型的文字作为ANYARRAY时的内核转储
- 修复查询包含一个返回多个行的SQL函数时，AFTER触发器的错误处理 (Tom)
- 修复 `ALTER TABLE ... TYPE` 以便为 `USING` 子句重新检查 `NOT NULL` (Tom)
- 修复 `string_to_array()` ，以便为单独的字符串处理重叠的匹配

  例如， `string_to_array('123xx456xxx789', 'xx')` 。
- 为psql的 `\d` 命令修复模式匹配的极端情况
- 修复/contrib/ltree中的索引损坏错误 (Teodor)
- 修复ecpg中的很多鲁棒性 (Joachim Wieland)
- 修复/contrib/dbmirror中的反斜杠逃逸
- 修复Win32平台上统计收集的稳定性 (Tom, Andrew)
- 修复AIX和Intel编译器 (Tom)

E.144. 版本 8.0.8

> 发布日期: 2006-05-23

这个版本包含各种自8.0.7以来的修复，包括对极其严重安全问题的修复。 关于8.0主版本的新特性信息，请参阅Section E.152。

E.144.1. 迁移到版本 8.0.8

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

针对CVE-2006-2313中描述的SQL注入攻击的完全安全和CVE-2006-2314 可能需要在应用代码中改变。如果你有应用嵌入了不可信的字符串到SQL命令中， 那么你应该尽快检查他们，以确保他们使用的是推荐的逃逸技术。 在大多数情况下，应用应该使用库或驱动（如libpq的`PQescapeStringConn()`）提供的子程序执行字符串逃逸， 而不是依赖于*ad hoc*代码执行逃逸。

E.144.2. 修改列表

- 更改服务器以拒绝在所有情况下无效编码的多字节字符 (Tatsuo, Tom)

  PostgreSQL已经朝这个方向发展了一段时间了， 检查现在一致的应用到所有编码和所有文本输入，并且现在总是错误而不仅仅是警告。 这个修改防御了CVE-2006-2313中描述的类型的SQL注入攻击。

- 拒绝在字符串文本中不安全的使用`\'`

  作为服务器端防御CVE-2006-2314中描述的类型的SQL注入攻击， 服务器现在只接受`''`不接受`\'`作为SQL字符串文本中ASCII单引号的表示。缺省的，仅当`client_encoding`设置为仅客户端的编码时(SJIS, BIG5, GBK, GB18030, 或 UHC), 拒绝`\'`，这是SQL注入有可能会发生的情况。 一个新的配置参数`backslash_quote`可以用来在需要时调整这个行为。 请注意，针对CVE-2006-2314的完全安全可能需要客户端侧的改变； `backslash_quote`的目的部分是为了让不安全的客户端是不安全的显而易见。

- 修改libpq的字符串逃逸例程， 意识到编码注意事项和`standard_conforming_strings`

  这修复了使用libpq的应用在CVE-2006-2313和CVE-2006-2314 中描述的安全问题，并且也预防了转换到SQL标准字符串文字语法的计划。 使用多个并发的PostgreSQL连接的应用应该迁移到`PQescapeStringConn()`和`PQescapeByteaConn()`， 以确保在每个数据库连接中使用的设置正确的做了逃逸。 "手动"做字符串逃逸的应用应该被修改为依赖库例程。

- 修复一些不正确的编码转换函数

  `win1251_to_iso`，`alt_to_iso`，`euc_tw_to_big5`，`euc_tw_to_mic`，`mic_to_euc_tw` 都中断了改变范围。

- 清理字符串中仅剩的 `\'` 的使用 (Bruce, Jan)
- 修复导致OR'd索引扫描有时丢失它们应该返回的行的bug
- 修复btree索引已经被截断时的WAL重放
- 为包含 `|` 的模式修复 `SIMILAR TO` (Tom)
- 修复 `SELECT INTO` 和 `CREATE TABLE AS`， 以在缺省表空间中创建表，而不是在基本目录中 (Kris Jurka)
- 修复服务器，以正确的使用自定义的DH SSL参数 (Michael Fuhr)
- 字符Intel Macs中的Bonjour (Ashley Clark)
- 修复各种小的内存泄露
- 修复在一些Win32系统上的口令提示问题 (Robert Kinberg)

E.145. 版本 8.0.7

> 发布日期: 2006-02-14

这个版本包含各种自8.0.6以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.145.1. 迁移到版本 8.0.7

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.6的版本升级而来， 那么请参阅8.0.6的版本声明。

E.145.2. 修改列表

- 修复 `SET SESSION AUTHORIZATION` 中潜在的崩溃 (CVE-2006-0553)

  如果服务器已经启用断言编译了（这不是缺省的），那么一个未授权的用户会导致服务器进程崩溃， 导致临时拒绝对其他用户的服务。感谢Akio Ishida报告这个问题。

- 修复自动插入的行中行可见性逻辑的bug (Tom)

  在少数情况下，一个通过当前命令插入的行会显示为早已有效了， 而它不应该显示为这样。修复在8.0.4、7.4.9和7.3.11版本中创建的错误。

- 修复pg_clog和pg_subtrans文件创建期间可能会导致"文件早已存在" 错误的竞态条件 (Tom)

- 修复缓存失效信息正好在错误的时间到达时，可能会导致崩溃的情况 (Tom)

- 为预备语句中的 `UNKNOWN` 参数适当的检查 `DOMAIN` 约束 (Neil)

- 确保 `ALTER COLUMN TYPE` 以正确的顺序处理 `FOREIGN KEY` 、 `UNIQUE` 和 `PRIMARY KEY` 约束 (Nakano Yoshihisa)

- 修复以允许恢复转储有交叉模式引用自定义操作符或操作符类 (Tom)

- 允许pg_restore在 `COPY` 失败之后正确的继续； 以前它尝试将剩余的 `COPY` 数据当做SQL命令 (Stephen Frost)

- 当没有指定数据目录时，修复pg_ctl `unregister` 崩溃 (Magnus)

- 修复AMD64和PPC上的ecpg崩溃 (Neil)

- 如果错误发生在PL/python中的参数传递期间，正确的恢复 (Neil)

- 修复PL/perl处理Win32上的区域设置，以匹配后端 (Andrew)
- 修复`log_min_messages`设置为`DEBUG3`或在Win32上的`postgresql.conf`中时的崩溃 (Bruce)
- 修复Win32、Cygwin、OS X、AIX的pgxs `-L` 库路径声明 (Bruce)
- 当检查Win32管理员权限时检查是否启用了SID (Magnus)
- 适当的拒绝超出范围的日期输入 (Kris Jurka)
- 在配置期间测试`finite`和`isinf`的存在的可移植性修复 (Tom)

E.146. 版本 8.0.6

> 发布日期: 2006-01-09

这个版本包含各种自8.0.5以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.146.1. 迁移到版本 8.0.6

运行8.0.X的用户不需要转储/恢复。不过，如果你是从一个早于8.0.3的版本升级而来， 那么请参阅8.0.3的版本声明。另外，如果你受到本地或者下面描述的plperl 问题的影响，你可能需要在升级之后在文本字段上 `REINDEX` 索引。

E.146.2. 修改列表

- 修复Windows编码，如果在ShmemBackendArray中没有更多的空间时，主进程将继续而不是退出 (Magnus)

  如果过多的连接请求紧密的到达，以前的行为将导致拒绝服务的情况。 这只应用到Windows端口。

- 修复在8.0中引进的错误，可能允许ReadBuffer返回一个早已使用过的页当做新页， 潜在的导致丢失最近提交的数据 (Tom)

- 修复在一个事务外面或一个失败的事务中发出的控制级别的描述信息 (Tom)

- 为认为不同字符组合相等的环境修复字符串比较，比如Hungarian (Tom)

  这可能需要 `REINDEX` 来修复在文本字段上现有的索引。

- 在主进程启动期间设置本地环境变量，以确保plperl稍后不会更改本地环境

  这修复了一个postmaster启动时的环境变量声明和initdb 声明的不同时发生的问题。在这个条件下，任何对plperl 的使用都有可能导致损坏索引。如果你遇到了这个问题，你可能需要 `REINDEX` 以修复在文本字段上现有的索引。

- 允许安装目录更灵活的重定位 (Tom)

  以前的版本只在所有安装目录路径都相同（除了最后一个组件）时，才支持重定位。

- 修复strpos()中长期存在的bug和在某些很少使用的Asian多字节字符设置中的正则表达式处理 (Tatsuo)

- 对返回 `RECORD` 的函数的各种修复 (Tom)
- 修复 `/contrib/pgcrypto` gen_salt中的bug， 该错误导致它没有为MD5和XDES算法使用所有可用的盐空间 (Marko Kreen, Solar Designer)

  Blowfish和标准DES的盐没有受到影响。
- 修复 `/contrib/dblink` ， 以在指定的字段数量和查询实际返回的数量不同时抛出一个错误，而不是崩溃 (Joe)

E.147. 版本 8.0.5

> 发布日期: 2005-12-12

这个版本包含各种自8.0.4以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.147.1. 迁移到版本 8.0.5

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.3的版本升级而来， 那么请参阅8.0.3的版本声明。

E.147.2. 修改列表

- 修复事务日志管理中的竞态条件

  有一个狭窄的窗口，I/O操作可以为错误的页面初始化，导致断言失败或数据损坏。

- 修复从错误中恢复之后的bgwriter问题 (Tom)

  发现后端写入器在写入错误之后泄露缓存针。虽然不是致命的， 但是可能会导致稍后VACUUM命令奇怪的堵塞。

- 阻止当前事务早已中止而客户端发送捆绑的协议信息时的失败
- `/contrib/ltree` 修复 (Teodor)
- AIX和HPUX编译修复 (Tom)
- 在Windows NO_SYSTEM_RESOURCES错误之后重新尝试文件读取和写入 (Qingqing Zhou)
- 修复 `log_line_prefix` 包含 `%i` 时的间发故障
- 修复Windows上psql处理长脚本时的性能问题 (Merlin Moncure)
- 修复丢失的 `pg_group` 平台文件的更新
- 修复外连接中长期存在的规划错误

  这个bug有时导致虚假的错误"RIGHT JOIN只被可合并连接的连接条件支持"。

- 推迟时区初始化直到 `postmaster.pid` 创建之后

  这避免了混乱的启动脚本预期pid文件迅速出现。

- 当删除一个表时，阻止pg_autovacuum中的内核转储
- 修复整行引用(`foo.*`)子查询结果的问题

E.148. 版本 8.0.4

> 发布日期: 2005-10-04

这个版本包含各种自8.0.3以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.148.1. 迁移到版本 8.0.4

运行8.0.X的用户不需要转储/恢复。不过， 如果你是从一个早于8.0.3的版本升级而来， 那么请参阅8.0.3的版本声明。

E.148.2. 修改列表

- 修复允许 `VACUUM` 删除 `ctid` 链太快的错误， 并且在跟随 `ctid` 连接的代码中添加更多检查

  这修复了在非常少的情况下会导致崩溃的长期存在的问题。

- 修复使用多字节字符设置时， `CHAR()` 正确的填充空格到指定的长度 (Yoshiyuki Asaba)

  在以前的版本中， `CHAR()` 的填充是不正确的， 因为它只填充到指定数量的字节，而不考虑存储多少个字符。

- 在提交 `CREATE DATABASE` 之前强制一个检查点

  这应该修复了崩溃发生在 `CREATE DATABASE` 之后不久时的最近的 "index is not a btree"失败的报告。

- 修复 `COPY` 中的只读事务的意义上的测试

  该代码以前禁止 `COPY TO` ， 而它应该禁止 `COPY FROM` 。

- 处理 `COPY` CSV模式输入中连续嵌入的新行

- 为接近年的结尾的日期修复 `date_trunc(week)`

- 修复子句上只引用内侧关系的外连接的规划问题

- 更深层的修复 `x FULL JOIN y ON true` 的极端情况

- 修复过分优化 `x IN (SELECT DISTINCT ...)` 和相关的情况

- 修复由于未经深思熟虑"fuzzy"花费比较而使用小的 `LIMIT` 值的查询的错误规划

- 让 `array_in` 和 `array_recv` 更偏向于验证它们的OID参数
- 修复查询中丢失的行，像 `UPDATE a=... WHERE a...` with GiST index on column `a`
- 提高日期时间分析的鲁棒性
- 改善部分写入WAL页的检查
- 提高启用SSL时的信号处理的鲁棒性
- 改善MIPS和M68K自旋锁的代码
- 在主进程启动期间不要尝试打开多于 `max_files_per_process` 个的文件
- 各种内存泄露修复
- 各种可移植性改善
- 更新时区数据文件
- 改善Windows上DDL加载失败的处理
- 改善Windows上随机数的生成
- 让 `psql -f filename` 在打开文件失败时返回一个非零的退出代码
- 修改pg_dump以更可靠的处理非继承的检查约束
- 修复Windows上pg_restore中的口令提示
- 修复PL/pgSQL，当变量是通过引用传递类型时，正确的处理 `var := var`
- 修复PL/Perl `%_SHARED`，以便它实际上共享
- 修复 `contrib/pg_autovacuum`，以允许睡眠间隔超过2000秒
- 更新 `contrib/tsearch2`，以使用当前的Snowball代码

E.149. 版本 8.0.3

> 发布日期: 2005-05-09

这个版本包含各种自8.0.2以来的修复，包括几个安全相关的问题。 关于8.0主版本的新特性信息，请参阅Section E.152。

E.149.1. 迁移到版本 8.0.3

运行8.0.X的用户不需要转储/恢复。不过， 它是一个处理在8.0.X系统目录的初始内容中找到的两个重要安全问题的可能的方式。 使用8.0.3的initdb的dump/initdb/reload序列将自动纠正这些问题。

较大的安全问题是内建字符设置编码转换函数可以通过未授权的用户从SQL命令中调用， 但是该函数不是设计来这样使用的，并且恶意选择参数时时不安全的。 该修复包括修改这些函数声明的参数列表，这样他们可以不再从SQL命令中调用。 （这不影响他们通过编码转换机制的正常使用。）

较小的问题是 `contrib/tsearch2` 模块创建了几个错误的声明为返回 `internal` 的函数，而他们并不接受 `internal` 参数。 这破坏了所有使用 `internal` 参数的函数的类型安全。

强烈建议所有安装修复这些错误，通过initdb或通过遵循下面给出的手动修复程序。 该错误至少允许未授权的数据库用户崩溃他们的服务器进程， 并且可能允许未授权的用户获得数据库超级用户的权限。

如果你不希望initdb，执行7.4.8版本声明 中相同的手动修复程序。

E.149.2. 修改列表

- 修改编码函数签名以阻止误用
- 修改 `contrib/tsearch2` ，以避免不安全的使用 `INTERNAL` 函数结果
- 防止 `record_out` 不正确的第二个参数
- 修复古老的竞态条件：允许一个事务因为某些目的（如SELECT FOR UPDATE） 被看做比其他目的提交的稍早些

  这是一个极其严重的错误，因为它会导致明显的数据不一致短暂的被应用可见。
- 修复关系扩展和VACUUM之间的竞态条件

这理论上会导致丢失一页最近插入的数据，尽管这种情况看起来有非常小的可能性。 没有已知的情况导致超过一个断言的失败。

- 修复 `TIME WITH TIME ZONE` 值的比较

  在使用了 `--enable-integer-datetimes` 配置开关的地方， 该比较代码是错误的。注意：如果你在 `TIME WITH TIME ZONE` 字段上有一个索引，它将需要在安装这个更新之后 `REINDEX`， 因为该修复纠正了字段值的排序顺序。

- 为 `TIME WITH TIME ZONE` 值修复 `EXTRACT(EPOCH)`
- 修复 `INTERVAL` 值中负的分数秒的错误显示

  这个错误只在使用了 `--enable-integer-datetimes` 配置开关时发生。

- 修复pg_dump以正确的转储包含 `%` 的触发器名字
- 更多 `contrib/intagg` 的64位修复
- 阻止返回 `RECORD` 的函数的不正确的优化
- 阻止在 `COALESCE(NULL,NULL)` 上的崩溃
- 为libpq修复Borland makefile
- 为 `timetz` 类型修复 `contrib/btree_gist` (Teodor)
- 让 `pg_ctl` 检查在 `postmaster.pid` 发现的PID，看看它是否仍然是一个活动的进程
- 修复由转储时间戳的添加引起的 `pg_dump` / `pg_restore` 问题
- 修复具体可持有游标和事务提交期间触发延迟的处罚器之间的相互作用
- 修复返回通过引用传递数据类型的SQL函数中的内存泄露

E.150. 版本 8.0.2

> 发布日期: 2005-04-07

这个版本包含各种自8.0.1以来的修复。关于8.0主版本的新特性信息， 请参阅Section E.152。

E.150.1. 迁移到版本 8.0.2

运行8.0.*的用户不需要转储/恢复。这个版本更新PostgreSQL 库的主版本号，所以如果他们不能找到正确编号的共享库， 可能需要重新连接一些用户应用。

E.150.2. 修改列表

- 增加所有接口库的主版本号 (Bruce)

  这应该在8.0.0中完成。它需要7.4.X版本的PostgreSQL客户端应用， 像psql，可以用在和8.0.X应用相同的机器上。 这可能需要重新连接使用这些库的用户应用。

- 添加仅Windows的 `wal_sync_method` 设置 `fsync_writethrough` (Magnus, Bruce)

  这个设置导致写入WAL时PostgreSQL透写任何磁盘驱动的写缓存。 这个行为以前称为 `fsync` ， 但是已经重命名了， 因为它在其他平台上表现的与 `fsync` 稍微有点不同。

- 在Windows上启用 `wal_sync_method` 设置 `open_datasync` ， 并让它成为那个平台的缺省 (Magnus, Bruce)

  因为缺省不再是 `fsync_writethrough` ， 如果磁盘驱动启用了写缓存， 那么在电源故障期间的数据丢失是有可能的。在Windows上要关闭写缓存， 从Device Manager中，选择该驱动属性，然后 `Policies` 。

- 新增缓存管理算法2Q替代ARC (Tom)

  这样做是为了避免在ARC上挂起US执照。 2Q代码比ARC对于某些工作负载可能有几个百分点的缓慢。 一个更好的缓存管理算法将在8.1中出现。

- 规划器调整以改善自由创建的表上的行为 (Tom)

- 允许plpgsql分配初始为 `NULL` 的数组元素 (Tom)

  以前该数组将保持 `NULL` ，但是现在它变成一个单元素数组。 在8.0中主要的SQL引擎改变为以这种方式处理一个空数组值的 `UPDATE` ， 但是plpgsql中相似的情况则被忽视了。

- 在plpython函数体中转换 `\r\n` 和 `\r` 为 `\n` (Michael Fuhr)

  这阻止了plpython代码是在Windows或Mac客户端上写的时的语法错误。

- 允许SPI游标处理返回回行的工具命令，比如 `EXPLAIN` (Tom)
- 修复 `ALTER TABLE SET WITHOUT OIDS` 之后的 `CLUSTER` 失败 (Tom)
- 减少 `ALTER TABLE ADD COLUMN` 的内存使用 (Neil)
- 修复 `ALTER LANGUAGE RENAME` (Tom)
- 记录仅Windows的pg_ctl的 `register` 和 `unregister` 选项 (Magnus)
- 确保后端关闭期间所做的操作都被统计收集器计数

  这是预计解决pg_autovacuum不够频繁的清理系统目录的报告— 没有告诉后端关闭期间临时表删除引起的目录删除。

- 为配置参数 `log_destination` 到 `eventlog` 修改Windows缺省 (Magnus)

  缺省的，运行在Windows上的服务器现在将发送日志输出到windows事件记录器而不是标准误差。

- 让Kerberos认证在Windows上工作 (Magnus)
- 允许没有被标记为拥有CREATEDB权限的超级用户 `ALTER DATABASE RENAME` (Tom)
- 为 `CREATE` 和 `DROP DATABASE` 修改WAL日志条目， 不指定绝对路径 (Tom)

  这允许不同机器上的时间点恢复可以有不同的数据库位置。请注意，`CREATE TABLESPACE` 在这种情况下仍然构成了风险。

- 修复后端带有创建了一个表并且在它上面打开了一个游标的打开的事务退出时的崩溃 (Tom)
- 修复 `array_map()`，这样它可以调用PL函数 (Tom)
- 几个 `contrib/tsearch2` 和 `contrib/btree_gist` 修复 (Teodor)
- 修复某些平台上一些 `contrib/pgcrypto` 函数的崩溃 (Marko Kreen)
- 为64位平台修复 `contrib/intagg` (Tom)
- 修复解析 `CREATE` 语句时的ecpg错误 (Michael)
- 绕开在ecpg上导致问题的powerpc和amd64上的gcc错误 (Christof Petig)
- 当本地环境是 `C` 时，不要使用 `upper()`、 `lower()` 和 `initcap()` 的对本地敏感的版本 (Bruce)

这允许这些函数在本地环境是 `C` 时，在为非7位的数据产生错误的平台上工作。

- 修复 `quote_ident()`，以便给匹配关键字的名字加引号 (Tom)
- 修复 `to_date()`，以在 `CC` 和 `YY` 字段都使用了时合理的行为 (Karel)
- 当给出一个0月的间隔时，阻止 `to_char(interval)` 失败 (Tom)
- 修复 `date_trunc('week')` 返回的错误星期

  `date_trunc('week')` 为一些年份中的一月份的前几天返回错误的年份。
- 为 `INET` 数据类型中的类 `D` 地址使用正确的缺省标记长度 (Tom)

E.151. 版本 8.0.1

> 发布日期: 2005-01-31

这个版本包含各种自8.0.0以来的修复，包括几个安全相关的问题。 关于8.0主版本的新特性信息，请参阅Section E.152。

E.151.1. 迁移到版本 8.0.1

运行8.0.0的用户不需要转储/恢复。

E.151.2. 修改列表

- 不允许非超级用户 `LOAD`

  在平台上，这将自动执行一个共享库的初始化函数（这至少包括Windows和基于ELF的Unix）， `LOAD` 可以用来让服务器执行任意的代码。感谢NGS Software报告这个问题。

- 检查聚集函数的创建者是否有权限执行指定的转换函数

  这个疏忽使它有可能绕开函数上的EXECUTE权限的拒绝。

- 修复contrib/intagg中安全和64位问题
- 添加需要的STRICT标记到某些贡献函数 (Kris Jurka)
- 避免plpgsql游标声明有太多的参数时的缓存溢出 (Neil)
- 让 `ALTER TABLE ADD COLUMN` 在所有情况下都强制域约束
- 为FULL和RIGHT外连接修复规划错误

  连接的结果错误的认为是和左侧输入的排序相同。这不止会传递错误排序的输出给用户， 还会在嵌套的合并连接情况下给出完全错误的回复。

- 改善分组的聚集查询的规划
- `ROLLBACK TO` `_savepoint_` 关闭自检查点以来创建的游标
- 修复Windows上不合适的后端栈大小
- 在Windows上避免SHGetSpecialFolderPath() (Magnus)
- 修复作为一个Windows服务运行pg_autovacuum时的一些问题 (Dave Page)

- pg_dump/pg_restore中的多个小bug修复
- 修复用于类型定义的命名结构的ecpg段错误 (Michael)

E.152. 版本 8.0.0

> 发布日期: 2005-01-19

E.152.1. 概述

在这个版本中主要的修改是：

Microsoft Windows 本地服务器

这是第一个作为一个服务器在Microsoft Windows® 上本地运行的PostgreSQL版本。 它可以作为一个Windows服务运行。这个版本支持基于NT的Windows版本， 像Windows 2000 SP4、Windows XP 和Windows 2003。老的版本像Windows 95、 Windows 98和Windows ME是不支持的， 因为这些操作系统没有支持PostgreSQL的基础结构。 已经创建了一个单独的安装项目简化在Windows上的安装— 参阅http://www.postgresql.org/ftp/win32/。

尽管测试贯穿了我们的发布周期，但是Windows端口并没有得到多年在生产环境中使用的益处， PostgreSQL在Unix平台上所拥有的。 因此它应该和一个新产品一样被看做相同级别的警告。

以前的版本需要Unix仿真工具箱Cygwin， 以便在Windows操作系统上运行服务器。PostgreSQL 在Windows上支持本地客户端已经很多年了。

检查点

检查点允许一个事务的特定部分在不影响该事务剩余部分的情况下中止。 以前的版本没有这种能力；没有办法从事务中的语句失败中恢复， 除了中止整个事务。这个特性对于需要从复杂事务中错误恢复的应用程序写是有价值的。

时间点恢复

在以前的版本中，没有办法从磁盘驱动的失败中恢复， 除了从一个以前的备份中还原或者使用一个备用应用服务器。 时间点恢复允许服务器的继续备份。你可以恢复到失败点或者到以前的一些事物。

表空间

表空间允许管理员为单独表、索引和数据库的存储选择不同的文件系统。 这提高了性能并且控制了磁盘空间的使用。 以前的版本为这样的任务使用initlocation和人工符号连接管理。

改进了缓冲区管理、`CHECKPOINT`、`VACUUM`

这个版本有一个更加智能的缓冲区替代策略，这将更好的利用可用的共享缓冲区和提高性能。 vacuum和检查点的性能影响也将减少。

更改字段类型

字段的数据类型现在可以使用 `ALTER TABLE` 来更改。

新增Perl服务器端语言

一个新的plperl服务器端语言版本现在支持一个持久的共享存储区域、 触发器、返回记录和记录的数组、和SPI调用访问数据库。

`COPY` 支持逗号分隔的值(CSV)

`COPY` 现在可以读取和写入逗号分隔的值的文件。 它也能灵活的解释非标准的引用和单独的字符。

E.152.2. 迁移到版本 8.0.0

想要从任何以前的版本迁移数据的用户需要使用pg_dump转储/恢复。

观察下面的不兼容性：

- 在 `READ COMMITTED` 序列化模式中， 不稳定的函数现在看到并发事务提交的结果一直到该函数中每个语句的开始， 而不是一直到调用该函数的交互命令的开始。
- 声明 `STABLE` 或 `IMMUTABLE` 的函数总是使用调用查询的快照， 并且因此看不到在该调用查询开始之后采取的动作的影响， 不管是在它们自己的事务中还是其他事务中。这样一个函数也必须是只读的， 意味着它不能使用任何SQL命令除了 `SELECT` 。
- 非延迟的 `AFTER` 触发器现在在触发器查询完成之后立即触发， 而不是基于当前交互命令的完成。这使得触发器查询发生在一个函数内部时有所不同： 触发器在函数继续它的下一个操作之前被调用。
- 服务器配置参数 `virtual_host` 和 `tcpip_socket` 已经用一个更加一般的参数 `listen_addresses` 替代了。另外， 服务器现在缺省监听 `localhost` ， 消除了在很多情节中对 `-i` 参数开关的需要。
- 服务器配置参数 `SortMem` 和 `VacuumMem` 已经被重命名为 `work_mem` 和 `maintenance_work_mem` ， 以更好的反映它们的使用。原始的名字在 `SET` 和 `SHOW` 中仍然支持。
- 服务器配置参数 `log_pid` 、 `log_timestamp` 和 `log_source_port` 已经用一个更加一般的参数 `log_line_prefix` 替代。
- 服务器配置参数 `syslog` 已经用一个更加合理的 `log_destination` 变量替代，以控制日志输出目的地。

- 服务器配置参数 `log_statement` 已经被改变了， 所以它可以有选择的只记录数据库修改或数据定义语句。 服务器配置参数 `log_duration` 现在只在 `log_statement` 输出该查询时输出。
- 服务器配置参数 `max_expr_depth` 已经用 `max_stack_depth` 替换了， `max_stack_depth` 测量物理堆栈大小，而不是表达式嵌套深度。 这帮助了防止会话由于递归函数引起堆栈溢出而终止。
- `length()` 函数不再计数 `CHAR(n)` 值中的尾随空白。
- 转换一个整数到 `BIT(N)` ，选取该整数的最右边的N位，不是像以前那样选择最左边的N位。
- 更新一个NULL数组值的元素或部分现在产生非空的数组结果， 也就是一个数组只包含分配到的位置。
- 数组输入值的语法检查已经被大大的收紧了。以前允许垃圾在一些奇怪的地方带有奇怪的结果， 现在导致一个错误。空字符串元素值现在必须被写为 `""` ， 而不是什么都不写。 还改变了关于围绕着数组元素的空格的行为：现在忽略尾随的空白， 为了对称带有前导的空白（也总是被忽略）。
- 现在检测整数算术操作符的溢出并且报告为一个错误。
- 与单字节 `"char"` 数据类型有关的算术操作符已经被删除了。
- `extract()` 函数（也称为 `date_part` ） 现在为BC日期返回适当的年份，它以前的返回比正确的年份少一。 该函数现在也为千年和世纪返回适当的值。
- `CIDR` 值现在必须有它们的非标记位为0。例如，我们不再允许 `204.248.199.1/31` 作为 `CIDR` 值。 这样的值应该决不被PostgreSQL接受并且将是拒绝的。
- `EXECUTE` 现在返回一个完成标签匹配执行的语句。
- psql的 `\copy` 命令现在读取或写入到查询的 `stdin/stdout` ，而不是psql的 `stdin/stdout` 。 以前的行为可以通过新的 `pstdin` / `pstdout` 参数访问。
- JDBC客户端接口已经从内核分配中删除， 现在托管在http://jdbc.postgresql.org。
- Tcl客户端接口也被删除了。有几个Tcl接口现在托管在 http://gborg.postgresql.org。
- 服务器现在使用它自己的时区数据库，而不是操作系统提供的那个。 这将在所有平台上提供一致的行为。在大多数情况下， 这应该在时区行为上没有明显的不同，除了 `SET` / `SHOW` `TimeZone` 使用的时区名可能与您的平台提供的不同。
- Configure的线程选项不再需要用户运行测试或编辑配置文件； 线程选项现在是自动检测的。
- 请注意，表空间已经实现，initlocation已经被删除。

- 用户定义的GiST索引的API已经被改变了。 Union和PickSplit方法现在传递一个指针到一个特殊 `GistEntryVector` 结构，而不是 `bytea` 。

E.152.3. 废弃的特性

PostgreSQL行为的一些方面已经被认定为是次优的。 为了向后兼容，这些在8.0中还没有被删除，但是他们已经被认为是废弃的， 并且将在下一个主版本中删除。

- 8.1版本将为间隔删除 `to_char()` 函数。
- 服务器现在警告传递到 `oid` / `float4` / `float8` 数据类型的空字符串，但是仍然像以前一样将它们解释为0。 在下一个主版本中，这些数据类型的空字符串将被认为是无效的输入。
- 缺省的，在PostgreSQL 8.0和以前的版本中表是带有 `OID` 创建的。在下一个版本中，将不会是这样情况了： 要创建一个包含 `OID` 的表，必须指定 `WITH OIDS` 子句， 或者必须设置 `default_with_oids` 配置参数。 如果用户的表需要OID与PostgreSQL的未来版本兼容， 那么鼓励用户明确的指定 `WITH OIDS` 。

E.152.4. 修改列表

下面你将看到一个版本8.0和以前的主版本之间的修改的清单。

E.152.4.1. 性能提升

- 支持交叉数据类型索引的使用 (Tom)

  在这个修改以前，如果数据类型不正好匹配那么许多查询不使用索引。 这次改进使得索引的使用更加直观和一致。

- 新的提高缓存的缓冲区置换策略 (Jan)

  以前的版本使用一个最近最少使用的(LRU)缓存在内存中保存最近引用的页面。 LRU算法不考虑一个特定缓存条目被访问的次数，所以大表扫描可能会占用有用的缓存页面。 新的缓存算法使用四个单独的列表追踪最近使用的和最频繁使用的缓存页面， 并且动态的基于工作负载优化他们的替换。这会导致更加有效的使用共享的缓冲区缓存。 在以前测试过共享缓冲区大小的管理员应该用这个新的缓存替换策略重新测试。

- 添加子进程定期的写脏的缓冲区，以减少检查点的写入 (Jan)

  在以前的版本中，检查点进程，每几分钟运行一次， 将所有的脏缓冲区写到操作系统的缓冲区缓存，然后刷新所有脏的操作系统缓冲区到磁盘。 这导致磁盘使用的一个定期的高峰经常伤害性能。 新的代码使用一个后端书写器以一个平稳的速度慢慢的磁盘写入，

这样检查点有少得多的脏页面写入到磁盘。另外，新的代码不发出一个全局的 `sync()` 调用，但是只 `fsync()` 自最后一个检查点以来的文件写入。 这会提高性能并最小化检查点期间的降级。

- 添加延长vacuum以减少性能影响的能力 (Jan)

  在忙碌的系统上， `VACUUM` 执行许多I/O请求， 这会影响其他用户的性能。这个版本允许你放缓 `VACUUM` 的速度以减少它对其他用户的影响，虽然这样会增加 `VACUUM` 的总的存续时间。

- 为重复的键提高B-tree索引性能 (Dmitry Tkach, Tom)

  当许多重复的值存在于索引当中时，这改进了扫描索引的方式。

- 在规划时使用动态生成的表大小估计 (Tom)

  以前规划器估算表的大小使用最后一个 `VACUUM` 或 `ANALYZE` 看到的值，都是关于物理表大小（页数）和行数。 现在，当前物理表大小从内核中获得，行数是通过表大小乘以最后一个 `VACUUM` 或 `ANALYZE` 看到的行密度 （每页的行数）估计的。这会产生更加可靠的估计， 以防表大小自最后一个内务工作命令之后发生了重大的变化。

- 用 `OR` 子句提高索引的使用 (Tom)

  这允许优化器在语句中使用索引，带有许多过去没有被索引的OR子句。 也可以在第一个字段是指定的、第二个字段是一个 `OR` 子句的一部分的地方使用多字段索引。

- 提高部分索引子句的匹配 (Tom)

  服务器现在在包含复杂 `WHERE` 子句的查询中使用部分索引更加智能。

- 提高GEQO优化器的性能 (Tom)

  GEQO优化器用来规划包含许多表（缺省的，12个或更多）的查询。 这个版本加速了分析查询的方式，以减少花费在优化上的时间。

- 其他优化器改进

  这里没有列出所有小的改进，但是很多特殊情况比以前的版本运行的更好了。

- 提高C函数的查找速度 (Tom)

  这个版本为动态加载的C函数使用一个哈希表查找信息。 这提高了它们的速度，所以它们执行起来就和服务器可执行的内建函数一样快。

- 添加类型特定的 `ANALYZE` 统计能力 (Mark Cave-Ayland)

  这个特性允许在为非标准的数据类型产生统计信息时有更大的灵活性。

- `ANALYZE` 现在为表达式索引收集统计信息 (Tom)

表达式索引（也称作功能性索引）允许用户不只是索引字段，还有表达式的结果和函数调用。 用这个版本，优化器可以收集和使用关于表达式索引内容的统计信息。 这可以大大的提高表达式索引相关的查询的规划的质量。

- 为 `ANALYZE` 新增两阶段抽样方法 (Manfred Koizar)

  当有效行的密度与表的不同区域的密度非常不同时，这给出了更好的统计。

- 加速 `TRUNCATE` (Tom)

  这找回了一些在7.4中观察到的性能丢失，而仍然保持 `TRUNCATE` 事务安全。

E.152.4.2. 服务器的变化

- 添加WAL文件归档和时间点恢复 (Simon Riggs)
- 添加表空间，这样管理员可以控制磁盘布局 (Gavin)
- 添加一个内建的日志循环程序 (Andreas Pflug)

  现在可以不用依赖于syslog或外部日志循环程序就可以方便的记录服务器信息了。

- 添加新的只读服务器配置参数以显示服务器兼容的时间设置： `block_size` 、 `integer_datetimes` 、 `max_function_args` 、 `max_identifier_length` 、 `max_index_keys` (Joe)
- 让 `sameuser` 、 `samegroup` 和 `all` 的引用删除 `pg_hba.conf` 中这些术语的特殊含义 (Andrew)
- 在缺省的 `pg_hba.conf` 中为 `localhost` 使用更加清楚的IPv6名字 `::1/128` (Andrew)
- 在 `pg_hba.conf` 示例中使用CIDR格式 (Andrew)
- 重命名服务器配置参数 `SortMem` 和 `VacuumMem` 为 `work_mem` 和 `maintenance_work_mem` （仍然支持老的名字） (Tom)

  这个修改是用来弄清楚批量操作，比如用 `maintenance_work_mem` 创建的索引和外键，而 `work_mem` 是在查询执行期间使用的工作空间。

- 允许使用服务器配置 `log_disconnections` 记录会话断开 (Andrew)
- 添加新的服务器配置参数 `log_line_prefix` ， 以允许控制在每个日志行里发出的信息 (Andrew)

  可用的信息包括用户名、数据库名、远程IP地址和会话启动时间。

- 删除服务器配置参数 `log_pid` 、 `log_timestamp` 、 `log_source_port` ；功能性上被 `log_line_prefix` 取代 (Andrew)

- 用一个统一的 `listen_addresses` 参数替代 `virtual_host` 和 `tcpip_socket` 参数 (Andrew, Tom)

  `virtual_host` 只能指定要监听的单个IP地址。 `listen_addresses` 允许指定多个地址。

- 默认监听localhost，减少了在很多情况下对 `-i` 主进程开关的需要 (Andrew)

  监听localhost (`127.0.0.1`)不会打开新的安全漏洞， 但是允许像Windows和JDBC那样配置，不支持本地套接字，不用特殊的调整就能工作。

- 删除 `syslog` 服务器配置参数，并且添加更多逻辑的 `log_destination` 变量来控制日志输出位置 (Magnus)

- 修改服务器配置参数 `log_statement` 接受值 `all` 、 `mod` 、 `ddl` 或 `none` ，以便选择记录哪条查询 (Bruce)

  这允许管理员只记录数据定义修改或只记录数据修改语句。

- 一些日志相关的配置参数以前可以通过普通用户调整，但是只能是在"更详细的"方向。现在对待它们更加严格了：只有超级用户可以设置它们。不过， 超级用户可以使用 `ALTER USER` 为非超级用户提供这些值的每用户设置。 还有，现在超级用户可以通过 `PGOPTIONS` 设置仅超级用户的配置参数值了。

- 允许配置文件放在数据目录的外面 (mlw)

  缺省的，配置文件保存在集群的顶级目录中。有了这个添加， 配置文件可以放置在该数据目录的外面，宽松了管理。

- 只有在第一次执行常量可以用于统计时计划预备查询 (Oliver Jowett)

  预备语句规划查询一次执行多次。虽然预备查询避免了每次使用时重新规划的开支， 但是规划的质量不知道在该查询中要使用的准确参数。在这个版本中， 未命名的预备语句的规划被推迟直到第一次执行，执行的实际参数值用作优化提示。 这允许使用行外传递参数，而不会导致性能损失。

- 允许 `DECLARE CURSOR` 接受参数 (Oliver Jowett)

  在 `Parse` 消息中用参数发出 `DECLARE CURSOR` 现在是有用的。 在 `Bind` 时发送参数值将被取代为游标查询的执行。

- 修复 `inet` 和 `cidr` 数据类型的哈希连接和聚集 (Tom)

  版本7.4正确的处理了混合 `inet` 和 `cidr` 值的哈希。 （这个bug在以前的版本中不存在，因为他们不会尝试哈希任何一个数据类型。）

- 让 `log_duration` 只在 `log_statement` 输出查询时输出 (Ed L.)

E.152.4.3. 查询的变化

- 添加了保存点（嵌套的事务） (Alvaro)
- 现在接受不支持的隔离级别了并且提升为最近支持的级别 (Peter)

  SQL规范声明如果一个数据库不支持一个特定的隔离级别，它应该使用下一个更加限制的级别。 这个修改符合那个建议。

- 允许 `BEGIN WORK` 像 `START TRANSACTION` 那样指定事务隔离级别 (Bruce)
- 修复规则产生的查询类型和原始提交的查询类型不同时的表的权限检查 (Tom)
- 实现美元的引用以简化单引号的使用 (Andrew, Tom, David Fetter)

  在以前的版本中，因为单引号必须被用来引用一个函数体， 在函数文本的内部使用单引号需要使用两个单引号或其他易于出错的符号。 在这个版本中，我们添加了使用“美元引用”来引用一个文本块的能力。 在不同的嵌套界别使用不同的引用分隔符的能力大大的简化了恰当引用的工作， 尤其是在复杂的函数中。美元引用可以用在任何需要引用文本的地方。

- 让 `CASE val WHEN compval1 THEN ...` 只评估 `val` 一次 (Tom)

  `CASE` 不再多次评估测试的表达式。这在表达式是复杂的或不稳定时有益。

- 在计算一个聚集查询的目标列表之前测试 `HAVING` (Tom)

  修复不正确的失败案例，
  如 SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0 。 这应该能工作但是以前可能会有被零除而失败。

- 用 `max_stack_depth` 参数替换 `max_expr_depth` 参数， 以堆栈大小的千字节测量 (Tom)

  这给出了一个相当严密的防御，针对由于逃逸递归函数引起的崩溃。 代替测量表达式嵌套的深度，我们现在直接测量执行堆栈的大小。

- 允许任意行的表达式 (Tom)

  这个版本允许SQL表达式包含任意的符合类型，也就是说，行值。 也允许函数更容易的接受行作为参数和返回行值。

- 允许 `LIKE` / `ILIKE` 在行和子查询比较中用作操作符 (Fabien Coelho)
- 避免特定于语言环境的情况下在标识符和关键字中转换基本的ASCII字符 (Tom)

  这解决了单词包含 `I` 和 `i` 的识别编码的"Turkish 问题"。 在7位ASCII设置之外折叠字符仍然是环境敏感的。

- 改进语法错误报告 (Fabien, Tom)

  语法错误报告比以前更加有用。

- 修改 `EXECUTE`，返回一个完成标签匹配执行语句 (Kris Jurka)

  以前的版本为任何 `EXECUTE` 调用返回一个 `EXECUTE` 标签。 在这个版本中，返回的标签将反应执行的命令。

- 避免在规则列表中发出 `NATURAL CROSS JOIN` (Tom)

  这样一个条款在逻辑上说不通，但是在某些情况下该规则反编译以前生成的这种语法。

E.152.4.4. 对象操作的变化

- 为cast、转换、语言、操作符类和大对象添加 `COMMENT ON` (Christopher)
- 添加新的服务器配置参数 `default_with_oids`， 控制表缺省是不是带有 `OID` 创建 (Neil)

  这允许管理员控制 `CREATE TABLE` 命令缺省是否带有 `OID` 字段创建表。（注意：目前 `default_with_oids` 的出厂默认设置是 `TRUE`， 但是该缺省将在将来的版本中变成 `FALSE` 。）

- 添加 `WITH` / `WITHOUT OIDS` 子句到 `CREATE TABLE AS` (Neil)
- 允许 `ALTER TABLE DROP COLUMN` 删除 `OID` 字段 （ `ALTER TABLE SET WITHOUT OIDS` 仍然工作） (Tom)
- 允许复合类型作为表字段 (Tom)
- 允许 `ALTER ... ADD COLUMN` 带有缺省和 `NOT NULL` 约束； 在每个SQL规范上工作 (Rod)

  `ADD COLUMN` 创建一个最初不是填充为NULL的字段现在是可能的了， 但是会带有一个指定的缺省值。

- 添加 `ALTER COLUMN TYPE` 改变字段的类型 (Rod)

  现在不用删除然后重新添加该字段来修改一个字段的数据类型是可能的了。

- 允许多个 `ALTER` 在一个 `ALTER TABLE` 命令中动作 (Rod)

  这对于重写表的 `ALTER` 命令尤其有用（包括 `ALTER COLUMN TYPE` 和带有一个缺省的 `ADD COLUMN` ）。通过一起分组 `ALTER` 命令， 该表只需要重写一次。

- 允许 `ALTER TABLE` 添加 `SERIAL` 字段 (Tom)

  这来自为新的字段指定缺省的新能力。

- 允许修改聚集、转换、数据库、函数、操作符、操作符类、模式、类型和表空间的所有者 (Christopher, Euler Taveira de Oliveira)

  以前这些需要直接修改系统表。

- 允许临时对象的创建局限于 `SECURITY DEFINER` 函数 (Sean Chittenden)
- 添加 `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)

  在该版本以前，没有方法清空一个自动群集的规范，除了修改系统表。

- 约束/索引/ `SERIAL` 名字现在是 `_table_column_type_` ， 附加上编号以保证在模式中的唯一性 (Tom)

  SQL规范声明这样的名字应该在一个模式中唯一。

- 添加 `pg_get_serial_sequence()` 来返回一个 `SERIAL` 字段的序列名 (Christopher)

  这允许自动化的脚本可靠的找到 `SERIAL` 序列名。

- 当主键/外键数据类型不匹配配需要昂贵的查找时警告
- 新增 `ALTER INDEX` 命令允许删除表空间之间的索引 (Gavin)
- 让 `ALTER TABLE OWNER` 修改依赖的序列所有权 (Alvaro)

E.152.4.5. 工具命令的变化

- 允许 `CREATE SCHEMA` 创建触发器、索引和序列 (Neil)
- 添加 `ALSO` 关键字到 `CREATE RULE` (Fabien Coelho)

  这允许 `ALSO` 被添加到规则创建，与 `INSTEAD` 规则对比。

- 添加 `NOWAIT` 选项到 `LOCK` (Tatsuo)

  这允许 `LOCK` 命令在它必须等待请求的锁时失败。

- 允许 `COPY` 读写逗号分隔值(CSV)的文件 (Andrew, Bruce)
- 如果 `COPY` 分隔符和NULL字符串冲突则产生错误 (Bruce)
- `GRANT` / `REVOKE` 行为更加符合SQL规范
- 避免 `CREATE INDEX` 和 `CHECKPOINT` 之间的锁冲突 (Tom)

  在7.3和7.4中，一个长期运行的B-tree索引建立会阻止并发的 `CHECKPOINT` 完成， 因此导致WAL膨胀，因为WAL日志不能重复利用。

- 数据库范围的 `ANALYZE` 不在表上持有锁 (Tom)

  这减少了其他后端想要在表上的排他锁时死锁的可能性。要获取这个修改的益处， 不要在一个事务块(`BEGIN` 块)中执行数据库范围的 `ANALYZE` ；必须能够为每个表提交和启动一个新事务。

- `REINDEX` 不再完全锁住索引的父表

  索引本身仍然是完全锁住的，但是该表的读者可以继续，如果他们不使用被重建的特定索引。

- 当一个用户重命名了时，擦除MD5用户口令 (Bruce)

  当通过MD5解密口令时，PostgreSQL 使用用户的名字作为盐。当改变了一个用户的名字时，盐将不再匹配存储的MD5口令， 所以存储的口令变得无用了。在这个版本中生成一个通知并且清除口令。 如果该用户能够使用一个口令登录，那么一个新的口令必须被分配。

- 为Windows新建pg_ctl `kill` 选项 (Andrew)

  Windows没有 `kill` 命令发送信号到后端，所以这个能力添加到了pg_ctl。

- 信息模式改善

- 添加 `--pwfile` 选项到initdb， 这样初始口令可以通过GUI工具设置 (Magnus)

- 检测initdb中的本地环境/编码错误匹配 (Peter)

- 添加 `register` 命令到pg_ctl， 以注册Windows操作系统服务 (Dave Page)

E.152.4.6. 数据类型和函数的变化

- 对复合类型（行类型）更加完整的支持 (Tom)

  复合值可以用在以前只能是标量值的许多地方。

- 作为错误拒绝非矩形的数组值 (Joe)

  以前， `array_in` 将默默地建立一个令人惊讶的结果。

- 现在检测到了整数算术运算符的溢出 (Tom)

- 与单字节 `"char"` 数据类型相关的算术操作符已经删除了。

  以前，解析器将在许多情况下选择这些操作符，而一个"不能选择一个操作符" 错误应该更加合适，比如 `null * null` 。如果你真的想要在一个 `"char"` 字段上做算术，你可以将它明确的转换为整型。

- 数组输入值的语法检查明显的严格了 (Joe)

  以前允许在旧的位置的带有旧的结果的垃圾现在导致一个 `ERROR` ， 例如，紧跟着右括号后面的非空白。

- 空字符串数组元素值现在必须写为 `""` ，而不是什么都不写 (Joe)

以前，两种书写空字符串元素值的方法都是允许的，但是现在要求用带有引号的空字符串。 在一些未来的版本中，什么都不出现的情况可以考虑为一个NULL元素值。

- 现在忽略数组元素的尾随空白 (Joe)

  以前忽略前导的空白，但是元素值和分隔符或右括号之间的尾随的空白是有效的。 现在尾随的空白也忽略了。

- 当下界不是1时，用明确的数组界限发出数组值 (Joe)
- 作为日期字符串接受 `YYYY-monthname-DD` (Tom)
- 让 `netmask` 和 `hostmask` 函数返回最大长度标记的长度 (Tom)
- 修改阶乘函数以返回 `numeric` (Gavin)

  返回的 `numeric` 允许阶乘函数为更大范围的输入值工作。

- `to_char` / `to_date()` 日期转换的改善 (Kurt Roeckx, Fabien Coelho)
- 让 `length()` 忽视 `CHAR(n)` 中尾随的空白 (Gavin)

  这个修改是为了提高一致性： `CHAR(n)` 数据中尾随的空白在语义上是无关紧要的， 所以它们不应该被 `length()` 计数。

- 警告空字符串被传递到 `OID` / `float4` / `float8` 数据类型 (Neil)

  8.1中将抛出一个错误。

- 允许前导或尾随的空白在 `int2` / `int4` / `int8` / `float4` / `float8` 输入例程中 (Neil)
- 在 `float4` / `float8` 中更好的支持IEEE `Infinity` 和 `NaN` 值 (Neil)

  这些应该在所有支持IEEE兼容浮点运算的平台上工作。

- 添加 `week` 选项到 `date_trunc()` (Robert Creager)
- 为 `1 BC` 修复 `to_char` （以前它返回 `1 AD` ）(Bruce)
- 为BC日期修复 `date_part(year)` （以前它返回的比正确的年份少1） (Bruce)
- 修复 `date_part()` 以返回正确的千年和世纪 (Fabien Coelho)

  在以前的版本中，与标准比较计算，世纪和千年的结果有一个错误的数值， 并且是以错误的年份开始的。

- 为了符合标准，添加 `ceiling()` 作为 `ceil()` 的一个别名， `power()` 作为 `pow()` 的一个别名 (Neil)
- 修改 `ln()` 、 `log()` 、 `power()` 和 `sqrt()` ， 就像SQL:2003中指定的那样，为确定的错误条件发出正确的 `SQLSTATE` 错误代码 (Neil)

- 像SQL:2003中定义的那样添加 `width_bucket()` 函数 (Neil)
- 添加 `generate_series()` 函数简化数字集的工作 (Joe)
- 修复 `upper/lower/initcap()` 函数以与多字节编码一起工作 (Tom)
- 添加布尔和按位整数 `AND` / `OR` 聚集 (Fabien Coelho)
- 新增会话信息函数，为客户端和服务器返回网络地址 (Sean Chittenden)
- 添加函数确定闭合路径的区域 (Sean Chittenden)
- 添加函数发送取消请求到其他后端 (Magnus)
- 添加 `datetime` 加 `datetime` 操作符 (Tom)

  相反的顺序， `datetime` 加 `interval` ，早已支持了， 但是SQL标准两种都需要。
- 转换一个整数到 `BIT(N)` ，选取该整数的最右侧的N位 (Tom)

  在以前的版本中，选择最左侧的N位，但是这被认为是没有帮助的， 更不用说与从位转换到整数的不一致。
- 需要 `CIDR` 值让所有非标记的位为0 (Kevin Brintnall)

E.152.4.7. 服务器端语言的变化

- 在 `READ COMMITTED` 序列化模式中， 不稳定的函数现在看到并发事务提交的结果直到该函数中每个语句的开始， 而不是直到调用该函数的交互命令的开始。
- 声明为 `STABLE` 或 `IMMUTABLE` 的函数总是使用调用查询的快照， 并且因此不能看到调用查询开始之后采取的动作的影响， 不管是在它们自己的事务中还是在其他事务中。这样一个函数必须也是只读的， 意味着它不能使用任何SQL命令，除了 `SELECT` 。 声明一个函数 `STABLE` 或 `IMMUTABLE` 而不是 `VOLATILE` 有相当大的性能增益。
- 非延迟的 `AFTER` 触发器现在在触发器查询完成之后立即触发， 而不是根据当前交互命令的完成。这在触发查询发生在一个函数中时有所不同： 触发器在该函数进行它的下一个操作之前被调用。例如， 如果一个函数插入了一个新行到一个表中，任何非延迟的外键检查在处理该函数之前发生。
- 允许函数参数用名字声明 (Dennis Björklund)

  这允许更好的记录函数。名字实际上是否做任何事情依赖于使用的特定的函数语言。
- 允许PL/pgSQL参数名在函数中引用 (Dennis Björklund)

  这基本上为每个命名的参数都创建了一个自动的别名。
- 在创建时做最少的PL/pgSQL函数的语法检查 (Tom)

这允许我们快速的捕捉简单的语法错误。

- 在PL/pgSQL中更加支持复合类型（行和记录变量）

  例如，现在可以将行类型变量作为单个变量传递到另一个函数了。

- PL/pgSQL变量的缺省变量现在可以引用以前声明的变量。

- 为循环改善PL/pgSQL的解析 (Tom)

  解析现在是通过 `".."` 的存在来驱动的，而不是 `FOR` 变量的数据类型。 这对于正确的函数来说没什么差别，但是当有一个错误时，会导致更容易理解的错误消息。

- 主要检修了PL/Perl服务器端语言 (Command Prompt, Andrew Dunstan)

- 在PL/Tcl中，SPI命令现在在子事务中运行。如果发生了一个错误， 该子事务被清理并且该错误被作为一个普通Tcl错误报告， 可以使用 `catch` 捕获。以前，捕获这样的错误是没有可能的。

- 在PL/pgSQL中接受 `ELSEIF` (Neil)

  以前PL/pgSQL只允许 `ELSIF` ，但是许多人习惯拼写 `ELSEIF` 关键字。

E.152.4.8. psql 的变化

- 改善psql关于数据库对象的信息显示 (Christopher)
- 允许psql在 `\du` 和 `\dg` 中显示组成员 (Markus Bertheau)
- 阻止psql `\dn` 显示临时模式 (Bruce)
- 允许psql为文件名处理波浪符用户扩展 (Zach Irmen)
- 允许psql显示提示，包括颜色，通过readline (Reece Hart, Chet Ramey)
- 让psql `\copy` 完全匹配 `COPY` 命令语法 (Tom)
- 显示语法错误的位置 (Fabien Coelho, Tom)
- 添加 `CLUSTER` 信息到psql `\d` 显示 (Bruce)
- 修改psql `\copy stdin/stdout` ， 以便从命令输入/输出中读取 (Bruce)
- 添加 `pstdin` / `pstdout` ，以便从psql 的 `stdin` / `stdout` 读取 (Mark Feit)
- 添加全局psql配置文件， `psqlrc.sample` (Bruce)

  这允许一个可以存储全局psql启动命令的中心文档。

- 让psql `\d+` 表明表是否有一个 `OID` 字段 (Neil)

- 在Windows上，当读取文件时使用二进制模式，这样control-Z不会被看做文件的结尾。
- 让 `\dn+` 显示模式的权限和描述 (Dennis Björklund)
- 提高选项卡完成支持 (Stefan Kaltenbrunn, Greg Sabino Mullane)
- 允许布尔设置使用大写或小写 (Michael Paesold)

E.152.4.9. pg_dump 的变化

- 使用依赖关系信息提高pg_dump的可靠性 (Tom)

  这应该解决了相关的对象有时会以错误的顺序转储的长期存在的问题。

- 如果可能，让pg_dump以字母顺序输出对象 (Tom)

  这将使得它更容易的识别出转储文件间的变化。

- 允许pg_restore忽略一些SQL错误 (Fabien Coelho)

  这使得pg_restore的行为类似于输出pg_dump 输出脚本到psql的结果。在大多数情况下， 忽略错误和向前进行是最有用的事情。 还添加了一个pg_restore选项给老的行为在遇到错误时退出。

- pg_restore `-l` 显示现在包括了对象的模式名
- 在pg_dump文本输出中新增开始/结束标记(Bruce)
- 在详细模式为pg_dump/pg_dumpall添加开始/结束时间 (Bruce)
- 在pg_dumpall中允许大部分pg_dump选项 (Christopher)
- 让pg_dump默认使用 `ALTER OWNER` 而不是 `SET SESSION AUTHORIZATION` (Christopher)

E.152.4.10. libpq的变化

- 让libpq的 `SIGPIPE` 处理线程安全 (Bruce)
- 添加 `PQmbdsplen()` ，返回一个字符的显示长度 (Tatsuo)
- 添加线程锁到SSL和Kerberos连接 (Manfred Spraul)
- 允许 `PQoidValue()` 、 `PQcmdTuples()` 和 `PQoidStatus()` 在 `EXECUTE` 命令上工作 (Neil)
- 添加 `PQserverVersion()` ，添加更多到服务器版本号的方便的访问 (Greg Sabino Mullane)
- 添加 `PQprepare/PQsendPrepared()` 函数支持预备语句， 没有必要指定它们的参数的数据类型 (Abhijit Menon-Sen)
- 许多ECPG改善，包括 `SET DESCRIPTOR` (Michael)

E.152.4.11. 源代码的变化

- 允许数据库服务器在Windows上本地运行 (Claudio, Magnus, Andrew)
- 为了Windows支持，Shell脚本命令转换为C版本 (Andrew)
- 创建一个扩展makefile框架 (Fabien Coelho, Peter)

  这简化了在源代码树的外面建立扩展的任务。
- 支持重新定位安装 (Bruce)

  安装文件的目录路径（比如 `/share` 目录）现在是和可执行文件的实际位置相关的， 所以一个安装树可以移动到另外一个地方而不用重新配置和重新建立。
- 使用 `--with-docdir` 选择文档的安装位置；也允许 `--infodir` (Peter)
- 添加 `--without-docdir` ，阻止文档的安装 (Peter)
- 升级到DocBook V4.2 SGML (Peter)
- 新增 `PostgreSQL` CVS标签 (Marc)

  这样做是为了使它更简单的组织管理它们自己的PostgreSQL CVS仓库。主仓库的文件版本邮票将不会通过检查进出一个复制的存储库而加上参数。 （File version stamps from the master repository will not get munged by checking into or out of a copied repository.）
- 明确锁定代码 (Manfred Koizar)
- 缓冲区管理器清理 (Neil)
- 从CPU自旋锁代码解耦平台测试 (Bruce, Tom)
- 在PA-RISC上为gcc添加在线测试和设置代码 (ViSolve, Tom)
- 改善i386自旋锁代码 (Manfred Spraul)
- 清理自旋锁汇编代码，以避免来自新的gcc版本的警告 (Tom)
- 从源码树中删除JDBC；JDBC现在是一个单独的项目
- 删除libpgtcl客户端接口；它现在是一个单独的项目
- 更准确的估计内存和文件描述符的使用 (Tom)
- 提高到Mac OS X启动脚本 (Ray A.)
- 新增 `fsync()` 测试程序 (Bruce)
- 主要文档的改进 (Neil, Peter)

- 删除pg_encoding;不再需要了
- 删除pg_id;不再需要了
- 删除initlocation;不再需要了
- 自动检测线程标志（不再手动检测） (Bruce)
- 使用Olson的公共域timezone库 (Magnus)
- 启用了线程，在Unixware上为后端可执行文件使用线程标志 (Bruce)

  Unixware在相同的可执行文件中不能混合线程的和非线程的对象文件， 所以每个都必须编译为线程的。

- psql现在使用一个flex产生的词法分析器处理命令字符串
- 重装整个后台使用的链表数据结构 (Neil)

  这通过允许附加列表和长度操作符更加高效提高了性能。

- 允许动态加载的模块创建它们自己的服务器配置参数 (Thomas Hallgren)
- 新增FAQ的Brazilian版本 (Euler Taveira de Oliveira)
- 添加French FAQ (Guillaume Lelarge)
- 为Windows登陆新增pgevent
- 让libpq和ECPG在OS X上作为适当共享的库建立 (Tom)

E.152.4.12. 贡献版的变化

- 彻底检查 `contrib/dblink` (Joe)
- `contrib/dbmirror` 改善 (Steven Singer)
- 新增 `contrib/xml2` (John Gray, Torchbox)
- 升级了 `contrib/mysql`
- 新增 `contrib/btree_gist` 的版本 (Teodor)
- 新增 `contrib/trgm` ，为PostgreSQL的三元模型匹配 (Teodor)
- 许多 `contrib/tsearch2` 的改善 (Teodor)
- 添加两倍metaphone到 `contrib/fuzzystrmatch` (Andrew)
- 允许 `contrib/pg_autovacuum` 作为Windows服务运行 (Dave Page)

- 添加函数到`contrib/dbsize` (Andreas Pflug)
- 删除`contrib/pg_logger`:被完整的登陆子进程代替了
- 删除`contrib/rserv`:被各种独立的项目代替了

E.153. 版本 7.4.30

> 发布日期: 2010-10-04

这个版本包含各种自7.4.29以来的修复。要想获得关于7.4主版本的新特性信息， 请参阅Section E.183。

这预计是PostgreSQL 7.4.X系列的最后一个版本。 推荐用户尽快更新到新版本。

E.153.1. 迁移到版本 7.4.30

运行7.4.X的用户不需要转储/恢复。不过，如果您是从一个早于7.4.26的版本升级而来， 那么请参考7.4.26的版本声明。

E.153.2. 修改列表

- 为每个在PL/Perl和PL/Tcl中调用的SQL userid使用一个单独的解释器 (Tom Lane)

  这个修改阻止可以由破坏稍后在同一个会话中另一个SQL用户身份执行的Perl或Tcl代码引起的安全问题 （例如，在一个 `SECURITY DEFINER` 函数中）。大多数脚本语言提供多种可能执行的方式， 比如重定义被目标函数调用的标准函数或操作符。没有这个修改， 任意拥有Perl或Tcl语言使用权限的SQL用户本质上都可以用目标函数的所有者的SQL权限做任何事情。

  这个修改的成本是Perl和Tcl函数之间有意的交流变得更加困难。为了提供一个安全舱口， PL/PerlU和PL/TclU函数继续每个会话只使用有一个解释器。这不认为是一个安全问题， 因为所有这样的函数都早已在数据库超级用户的信任级别执行了。

  有可能要求提供受信任的执行的第三方的过程语言有相似的安全问题。 我们建议为了关键性的安全目的，联系任何你依赖的PL的作者。

  感谢Tim Bunce指出这个问题 (CVE-2010-3433)。

- 阻止 `pg_get_expr()` 中可能的崩溃，通过不允许它被调用， 有一个争论是它尝试使用多个系统目录字段 (Heikki Linnakangas, Tom Lane)

- 修复"cannot handle unplanned sub-select"错误 (Tom Lane)

  当一个子查询包含一个连接别名引用扩大为一个表达式包含另一个子查询时会发生这个错误。

- 在写锁文件（包括`postmaster.pid`和套接字锁文件）时要小心的同步锁文件的内容 (Tom Lane)

  如果机器在主进程启动后很快就崩溃了，那么这个疏忽会导致锁文件内容损坏。 这会阻止随后启动主进程的尝试成功，直到手动的移除锁文件。

- 改善`contrib/dblink`处理含有删除的字段的表 (Tom Lane)

- 修复`contrib/dblink`中出现"duplicate connection name" 错误之后的连接漏洞 (Itagaki Takahiro)

- 更新基础构造和文档，以反应源代码从CVS迁移到了Git (Magnus Hagander and others)

E.154. 版本 7.4.29

> 发布日期: 2010-05-17

这个版本包含各种自7.4.28以来的修复。关于7.4主版本的新特性信息， 请参阅Section E.183。

PostgreSQL社区将在2010年7月份停止对7.4.X版本系列的更新。 建议用户尽快更新到最新的版本。

E.154.1. 迁移到版本 7.4.29

运行7.4.X的版本不需要转储/恢复。不过，如果你是从一个早于7.4.26的版本升级而来， 那么请参阅7.4.26的版本声明。

E.154.2. 修改列表

- 使用一个opmask应用到整个解释器，执行 `plperl` 中的限制条件， 而不是使用 `Safe.pm` (Tim Bunce, Andrew Dunstan)

  最近的发展使我们相信 `Safe.pm` 太不安全了，不能依赖它来使得 `plperl` 可以信赖。这个修改完全删除了 `Safe.pm` 的使用， 支持总是使用一个单独的带有操作码的解释器。这个修改令人愉快的副作用包括： 现在在 `plperl` 中以自然的方式使用Perl的 `strict` 编译指示是可能的了， Perl的 `$a` 和 `$b` 变量在排序例程中像预期的那样工作了， 并且函数编译更快了。 (CVE-2010-1169)

- 阻止PL/Tcl执行 `pltcl_modules` 中不可靠的代码 (Tom)

  PL/Tcl自动从数据库表中加载Tcl代码的特性可能会被特洛伊木马攻击利用， 因为没有谁可以创建或插入到那个表的限制。这个修改使该特性失效， 除非 `pltcl_modules` 属于超级用户。（不过，在表上的权限是不检查的， 所以实际上需要较少安全模块的表的安装仍然可以赋予合适的权限给受信任的非超级用户。） 还有，阻止加载代码到不受限制的"正常"Tcl解释器， 除非我们真正要执行一个 `pltclu` 函数。 (CVE-2010-1170)

- 不允许非特权的用户重置超级用户仅有的参数设置 (Alvaro)

  以前，如果一个非特权用户为他自己运行 `ALTER USER ... RESET ALL`， 或者为他拥有的一个数据库运行 `ALTER DATABASE ... RESET ALL`， 都将为该用户或数据库删除所有的特殊参数设置，即使其中有只支持超级用户可以修改的设置。 现在， `ALTER` 将只删除用户有权限修改的参数。

- 如果关闭发生在 `CONTEXT` 添加来记录条目时，避免后端关闭期间可能的崩溃 (Tom)

  在某些情况下，上下文打印功能可能会失败，因为当它要打印一个日志信息时， 当前的事务早已回滚了。

- 为现在的Perl版本更新pl/perl的 `ppport.h` (Andrew)

- 修复pl/python中的各种内存泄露 (Andreas Freund, Tom)

- 确保 `contrib/pgstattuple` 函数响应迅速的取消中断 (Tatsuhito Kasahara)

- 使服务器启动适当的处理 `shmget()` 为一个已经存在的共享内存段返回 `EINVAL` 的情况 (Tom)

  这个行为已经在BSD衍生的核心程序包括OS X上观察到。它导致一个完全误导的启动失败， 抱怨共享内存请求大小太大。

E.155. 版本 7.4.28

> 发布日期: 2010-03-15

这个版本包括各种自7.4.27以来的修复。关于7.4主版本的新特性的信息， 请参阅Section E.183。

PostgreSQL社区将在2010年7月份停止对7.4.X版本系列的更新。 建议用户尽快更新到最新的版本。

E.155.1. 迁移到版本 7.4.28

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.26的版本升级而来， 那么请查看7.4.26的版本声明。

E.155.2. 修改列表

- 添加新的配置参数 `ssl_renegotiation_limit`， 控制多长时间为SSL连接做一次会话秘钥重新协商 (Magnus)

  可以设置为0来完全禁用重新协商，如果使用了破损的SSL库可能会需要这个。 特别的，一些供应商提供暂时的CVE-2009-3555的补丁导致重新协商的尝试失败。

- 使得 `substring()` 的 `bit` 类型对待负的长度为意味着 "所有剩余的字符串" (Tom)

  以前的代码只对待-1为这种方式，并且会为其它负值生成一个无效的结果值， 有可能会导致崩溃 (CVE-2010-0442)

- 修复一些正则表达式匹配变慢的情况 (Tom)

- 当读取 `pg_hba.conf` 和相关的文件时，如果 `@` 出现在双引号标记的内部， 那么不要将 `@something` 当做文件包含查询；还有， 永远不要将 `@` 本身当做文件包含查询 (Tom)

  这阻止了当用户或者数据库名字以 `@` 开始时的不规律行为。 如果你需要包括一个文件的整个路径名（包括空格），你仍然可以这样做， 但是必须写 `@"/path to/file"` 而不是使双引号包含整个构造。

- 如果一个路径命名为 `pg_hba.conf` 和相关文件中的包含目标， 阻止某些平台上的无限循环 (Tom)

- 确保PL/Tcl完全初始化Tcl解释器 (Tom)

  现在知道的唯一情况是，如果使用Tcl 8.5或以后的版本，Tcl `clock` 命令会错误动作。

- 当太多的关键字段在`dblink_build_sql_*`函数中声明时， 阻止`contrib/dblink`的崩溃 (Rushabh Lathia, Joe Conway)

E.156. 版本 7.4.27

> 发布日期: 2009-12-14

这个版本包含各种自7.4.26以来的修复。关于7.4主版本的新特性的信息， 请参阅Section E.183。

E.156.1. 迁移到版本7.4.27

运行7.4.X的用户不需要转储/恢复。不过， 如果你是从一个早于7.4.26的版本升级而来， 那么请查看7.4.26的版本声明。

E.156.2. 修改列表

- 防御索引函数改变会话环境状态引起的间接安全威胁 (Gurjeet Singh, Tom)

  这个修改阻止早已不变的索引函数可能破坏超级用户的会话 (CVE-2009-4136)。

- 拒绝SSL证书在common name（CN）字段包含一个嵌入的空字节 (Magnus)

  这阻止了在SSL生效期间非故意的匹配一个证书到一个服务器或客户端名 (CVE-2009-4034)。

- 修复后端启动时缓存初始化期间可能的崩溃 (Tom)

- 阻止信号在不安全的时间打断 `VACUUM` (Alvaro)

  如果 `VACUUM FULL` 在它已经提交的元组动作之后取消，那么这个修复阻止了一个PANIC，还有如果普通 `VACUUM` 在表截断之后中断时的瞬态错误。

- 修复哈希表大小计算中由于整数溢出可能的崩溃 (Tom)

  这会在规划器估算哈希连接结果的尺寸非常大时发生。

- 修复 `inet` / `cidr` 比较中非常罕见的崩溃 (Chris Mikkelson)

- 修复PAM密码处理更加强健 (Tom)

  以前的代码都知道是在Linux `pam_krb5` PAM模块和Microsoft Active Directory 组合为域控制器时失败。也有可能会在其他地方有问题，因为它对于PAM堆传递哪个参数做了不正确的假设。

- 使得主进程忽略任意连接请求包中的 `application_name` 参数， 以提高与未来libpq版本的兼容性 (Tom)

E.157. 版本 7.4.26

> 发布日期: 2009-09-09

这个版本包含各种自7.4.25以来的修复。关于7.4主版本的新特性信息， 请参阅Section E.183

E.157.1. 迁移到版本7.4.26

运行7.4.X版本的用户不需要转储/恢复。不过，如果你在 `interval` 字段上有任何哈希索引，你必须在升级到7.4.26以后 `REINDEX` 它们。 还有，如果你从一个早于7.4.11的版本升级而来，请查看7.4.11的版本声明。

E.157.2. 修改列表

- 不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` 在安全定义函数里面 (Tom, Heikki)

  这包含了以前的补丁丢失的一个情况，不允许 `RESET ROLE` 和 `RESET SESSION AUTHORIZATION` 在安全定义函数里面。 (参阅 CVE-2007-6600)

- 修复子SELECT出现在外部级别聚合函数的参数中的处理 (Tom)
- 修复数据类型 `interval` 的哈希计算 (Tom)

  这纠正了间隔值上的哈希连接的错误结果。也改变了间隔字段上的哈希索引的内容。 如果有任何这样的索引，必须在升级之后 `REINDEX` 它们。

- 修复 `INTERVAL '``_x_` ms'中的 `_x_` 超过2百万，并且正在使用整数时间维度时的溢出 (Alex Hunsaker)
- 修复点到线段的距离的计算 (Tom)

  这导致一些几何操作符的不正确的结果。

- 修复 `money` 数据类型在货币数量没有小数点的环境中工作，如Japan (Itagaki Takahiro)
- 适当的圆整日期时间输入，如 `00:12:57.9999999999999999999999999999` (Tom)
- 修复GiST R-tree操作符类中页分裂点的欠佳的选择 (Teodor)
- 修复plperl初始化中的可移植性问题 (Andrew Dunstan)
- 改善libpq的代码从 `COPY FROM STDIN` 期间的错误中恢复的稳健性 (Tom)
- 避免包含readline和editline头文件的冲突，当两者的库都安装了时 (Zdenek Kotala)

E.158. 版本 7.4.25

> 发布日期: 2009-03-16

这个版本包含各种自7.4.24以来的修复。关于7.4主版本的新特性信息， 请查看Section E.183。

E.158.1. 迁移到版本 7.4.25

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.158.2. 修改列表

- 当编码转换失败时，阻止错误的递归崩溃 (Tom)

  这个修改扩展了在上两个主版本中相关错误情节的修复。 以前的修复正好适合原始的错误报告，但是我们现在意识到任何 编码转换函数抛出的错误都可能在尝试报告该错误时潜在的导致无限递归。 因此，如果我们发现我们已经进入递归的错误报告情形，解决方法是禁用翻译和编码转换， 并报告任何错误消息的原ASCII格式。 (CVE-2009-0922)

- 不允许 `CREATE CONVERSION` 用错误的编码指定转换函数 (Heikki)

  这防止了一个编码转换失败的可能的情形。 以前的修改是支持防范在相同地方的其他类型的失败。

- 修复当 `to_char()` 给出的格式代码不适合数据参数的类型时的内核转储 (Tom)

- 添加 `MUST` (Mauritius Island Summer Time)到已知的时区缩写的缺省列表 (Xavier Bugaud)

E.159. 版本 7.4.24

> 发布日期: 2009-02-02

这个版本包含各种自7.4.23以来的修复。关于7.4主版本的新特性信息， 请参考Section E.183。

E.159.1. 迁移到版本 7.4.24

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.159.2. 修改列表

- 改善URL在 `headline()` 函数中的处理 (Teodor)
- 改善 `headline()` 函数中超长标题的处理 (Teodor)
- 如果用错误的转换函数为两个指定的编码创建了一个编码转换， 那么阻止可能的断言失败或错误转换 (Tom, Heikki)
- 避免在 `VACUUM` 中锁定不必要的小表 (Heikki)
- 修复 `contrib/tsearch2` 的 `get_covers()` 函数中的未初始化的变量 (Teodor)
- 修复 `to_char()` 处理 `TH` 格式代码中的bug (Andreas Scherbaum)
- 使得所有的文档引用 `pgsql-bugs` 和/或 `pgsql-hackers` 是适当的，代替现在不用了的 `pgsql-ports` 和 `pgsql-patches` 邮件列表 (Tom)

E.160. 版本 7.4.23

> 发布日期: 2008-11-03

这个版本包含各种自7.4.22以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.160.1. 迁移到版本 7.4.23

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.160.2. 修改列表

- 修复当客户端编码不能表示本地化的错误消息时的后端崩溃 (Tom)

  我们以前解决过相似的问题，但是如果"character has no equivalent" 消息本身不能被转换时仍然会失败。该修复是为了在我们检测到这样一个情况时禁用本地化， 并且发送纯ASCII消息。

- 修复单个查询条目匹配文本的第一个单词时生成不正确的tsearch2标题 (Sushant Sinha)

- 当在一个 `--enable-integer-datetimes` 构造中使用一个非ISO日期类型时， 修复间隔值中分数秒的不恰当的显示 (Ron Mayer)

- 当传递的元组和元组描述符有不同的字段数量时， 确保 `SPI_getvalue` 和 `SPI_getbinval` 正确的行为 (Tom)

  当一个表有添加或删除的字段时，这个情形是正常的，但是这两个函数不能适当的处理它。 唯一可能的结果是一个不正确的错误提示。

- 修复ecpg的 `CREATE USER` 的语法分析 (Michael)

E.161. 版本 7.4.22

> 发布日期: 2008-09-22

这个版本包含各种自7.4.21以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.161.1. 迁移到版本 7.4.22

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.161.2. 修改列表

- 修复日期时间输入函数，当运行在64位的平台上时正确的检测整数溢出 (Tom)
- 提升向系统日志写入非常长的日志信息时的性能 (Tom)
- 修复`SELECT DISTINCT ON`查询上向后扫描一个游标中的bug (Tom)
- 修复规划器估算`GROUP BY`表达式产生的布尔结果总是在两个组里， 不管表达式的内容是什么 (Tom)

  这比普通`GROUP BY`估算确定的布尔测试（像`_col_ IS NULL`） 实质上来说更加精确。
- 改善未能发送一个SQL命令之后pg_dump和pg_restore 的错误报告 (Tom)

E.162. 版本 7.4.21

> 发布日期: 2008-06-12

这个版本包含一系列7.4.20的bug修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.162.1. 迁移到版本 7.4.21

运行7.4.X的用户不需要转储/恢复。不过， 如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.162.2. 修改列表

- 使 `pg_get_ruledef()` 将负的常量加入括号内 (Tom)

  在这个修复之前，视图或规则中的一个负的常量可能会被抛弃，说 `-42::integer` ， 这是不正确的：应该是 `(-42)::integer` ，因为操作符优先的规则。 通常这会有一点不同，但是它会与另外一个最近的路径相互作用导致PostgreSQL 拒绝一个有效的 `SELECT DISTINCT` 视图查询。因为这会导致pg_dump 的输出未能重载，所以被视为一个高优先级的修复。转储输出实际上不正确的发行版本是8.3.1和8.2.7。

E.163. 版本 7.4.20

> 发布日期: 从未发布

这个版本包含各种自7.4.19以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.163.1. 迁移到版本 7.4.20

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.163.2. 修改列表

- 修复ISO-8859-5和其他编码之间的转换，以处理Cyrillic "Yo"字符 （带有两个点的 `e` 和 `E` ） (Sergey Burladyan)
- 修复一些数据类型输入函数，他们允许未使用的字节在他们的结果中包含未初始化的、不可预期的值 (Tom)

  这会导致两个表面上一样的字面值不被认为相等的失败，导致分析器抱怨不匹配的 `ORDER BY` 和 `DISTINCT` 表达式。
- 修复正则表达式子字符串匹配中的极端情况 (`substring(``_string_` from `_pattern_`)) (Tom)

  这个问题发生在有一个完全匹配的模式的时候，但是用户已经指定了一个加括号的子表达式， 并且该子表达式没有得到匹配。一个例子是 `substring('foo' from 'foo(bar)?')` 。 这个应该返回NULL，因为 `(bar)` 不匹配， 但是它错误的返回了整个模式匹配（也就是， `foo` ）。
- 修复ecpg的 `PGTYPEStimestamp_sub()` 函数中不正确的结果 (Michael)
- 修复 `DatumGetBool` 宏使用gcc 4.3时不会失败 (Tom)

  这个问题影响"老式风格" (V0)的C函数返回布尔值。 该修复已经在8.3中，但是当时没有意识到需要回来修补它。
- 修复长期存在的 `LISTEN` / `NOTIFY` 竞态条件 (Tom)

  在少数情况下，刚刚执行了一个 `LISTEN` 的会话可能不会得到一个通知， 即使应该这样，因为并发事务执行 `NOTIFY` 会观察到提交以后。

该修复的一个副作用是一个执行了还未提交的 `LISTEN` 命令的事务将看不到任何在 `pg_listener` 中的行， 而它应该是能看到的；以前它也是能看到的。 这个行为从未有过这种或那种的记录，但是一些依赖于老的行为的应用是有可能出现的。

- 修复 `ORDER BY` 和 `GROUP BY` 中常数表达式的显示 (Tom)

  一个明确转换的常量会不正确的显示。这会导致，例如转储和重载期间视图定义损坏。

- 修复libpq以正确的处理COPY OUT期间的NOTICE消息 (Tom)

  这个故障只在一个用户定义的数据类型的输出程序发出一个NOTICE时可以观察到发生，但是不保证它不会由于其他原因发生。

E.164. 版本 7.4.19

> 发布日期: 2008-01-07

这个版本包含各种自7.4.18以来的修复，包括重大安全问题的修复。 关于7.4主版本的新特性的信息，请查阅Section E.183。

E.164.1. 迁移到版本 7.4.19

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.164.2. 修改列表

- 阻止索引中的函数用用户的权限执行 `VACUUM` 、 `ANALYZE` 等 (Tom)

  在索引表达式和部分索引判断中使用的函数在创建一个新的表条目时评估。 早就知道如果某人修改了一个不受信任的用户拥有的表，会造成特洛伊木马代码执行的风险。 （请注意，触发器、缺省、检查约束等，造成同样类型的风险。）但是索引中的函数造成额外的危险， 因为它们将被日常维护操作（如 `VACUUM FULL` ）执行， 而它们在超级用户账户下通常是自动执行的。例如， 一个不法用户通过设置一个特洛伊木马索引定义并等待下一个日常清理，可以使用超级用户权限执行代码。 该修复为标准的维护操作安排了(包括 `VACUUM` 、 `ANALYZE` 、 `REINDEX` 、 和 `CLUSTER`)作为表的所有者执行而不是调用的用户， 相同的权限切换机制早已为 `SECURITY DEFINER` 函数使用了。 为了阻止绕开这个安全措施， `SET SESSION AUTHORIZATION` 和 `SET ROLE` 现在禁止在 `SECURITY DEFINER` 内容中执行。 (CVE-2007-6600)

- 修复正则表达式包中的各种bug (Tom, Will Drewry)

  适当配置的正则表达式模式可能会导致崩溃、无限或接近无限循环、和/或大量的内存消耗， 所有这些都造成从不受信任的源接受正则表达式搜索模式的应用拒绝服务的风险。 (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- 要求使用 `/contrib/dblink` 的非超级用户只使用密码认证作为安全措施 (Joe)

  在7.4.18中出现的这个修复是不完全的，因为它只堵住了一些 `dblink` 函数的漏洞。 (CVE-2007-6601, CVE-2007-3278)

- 修复 `WHERE false AND var IN (SELECT ...)` 的某些情况下规划器失败 (Tom)

- 修复使用多字节数据库编码时 `translate()` 中潜在的崩溃 (Tom)

- 修复PL/Python在长的异常消息时不崩溃 (Alvaro)
- ecpg语法分析器修复 (Michael)
- 使 `contrib/tablefunc` 的 `crosstab()` 独立的将NULL rowid作为一个类别处理，而不是崩溃 (Joe)
- 修复 `tsvector` 和 `tsquery` 输出例程，以正确的逃逸反斜杠 (Teodor, Bruce)
- 修复 `to_tsvector()` 在大的输入字符串上的崩溃 (Teodor)
- 当重新生成 `configure` 脚本时，要求指定要使用的Autoconf的版本 (Peter)

  这只影响开发者和打包者。这个修改是为了阻止未经测验的Autoconf 和PostgreSQL版本的组合的意外使用。 如果你实在是想要使用一个不同的Autoconf版本， 可以移除版本校验，但是结果如何就是你的责任了。

E.165. 版本 7.4.18

> 发布日期: 2007-09-17

这个版本包含自7.4.17以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.165.1. 迁移到版本 7.4.18

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.165.2. 修改列表

- 当一个事务插入行然后在接近相同表中并发的 `VACUUM` 的结尾退出时， 阻止索引损坏 (Tom)
- 使得 `CREATE DOMAIN ... DEFAULT NULL` 适当的运行 (Tom)
- 修复SSL错误消息过多的记录 (Tom)
- 修复 `log_min_error_statement` 日志耗尽内存时的崩溃 (Tom)
- 阻止 `CLUSTER` 由于尝试处理其他会话的临时表引起的失败 (Alvaro)
- 要求使用 `/contrib/dblink` 的非超级用户只使用密码认证，作为一个安全措施 (Joe)

E.166. 版本 7.4.17

> 发布日期: 2007-04-23

这个版本包含自7.4.16以来的修复，包括一个安全修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.166.1. 迁移到版本 7.4.17

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.166.2. 修改列表

- 支持在 `search_path` 中的临时表模式有明确的位置， 并且禁用函数和操作符搜索它 (Tom)

  这需要允许安全定义函数设置真实的 `search_path` 的安全值。 没有他，无特权的SQL用户可以使用临时对象以安全定义函数的权限执行代码 (CVE-2007-2138)。 参阅 `CREATE FUNCTION` 获取更多信息。

- 修复 `/contrib/tsearch2` 崩溃 (Teodor)
- 修复 `VACUUM FULL` 如何处理 `UPDATE` 链条中潜在的数据损坏bug (Tom, Pavan Deolasee)
- 修复哈希索引扩大时的PANIC （在7.4.15引进的bug） (Tom)

E.167. 版本 7.4.16

> 发布日期: 2007-02-05

这个版本包含各种自7.4.15以来的修复，包括一个安全修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.167.1. 迁移到版本 7.4.16

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.167.2. 修改列表

- 删除允许连接的用户读取后端内存的安全漏洞 (Tom)

  弱点包括：抑制了SQL函数返回声明的数据类型的正常检查， 或改变了用于SQL函数中的表字段的数据类型 (CVE-2007-0555)。 这个错误可以轻易的被利用，导致后端崩溃，原则上， 可能被用来读取用户不应该可以访问的数据库内容。

- 修复由于选择了一个不可行的分裂点引起的btree索引页分裂失败的少见bug (Heikki Linnakangas)

- 修复由 `UNION` 触发的少见的Assert()崩溃 (Tom)

- 加强超过3字节长度的UTF8序列的多字节字符处理的安全 (Tom)

E.168. 版本 7.4.15

> 发布日期: 2007-01-08

这个版本包含各种自7.4.14以来的修复。关于7.4主版本的新特性的信息，请查阅Section E.183。

E.168.1. 迁移到版本 7.4.15

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来，请查看7.4.11的版本声明。

E.168.2. 修改列表

- 改善 `getaddrinfo()` 在AIX上的处理 (Tom)

  这修复了启动统计收集器的问题，除了别的之外。

- 修复 `VACUUM` 中的"failed to re-find parent key"错误 (Tom)
- 修复影响多个千兆字节的哈希索引的bug (Tom)
- 修复构造一个由多个空的元素组成的 `ARRAY[]` 时的错误 (Tom)
- 对于新的initdb安装， `to_number()` 和 `to_char(numeric)` 现在是 `STABLE` ， 不是 `IMMUTABLE` (Tom)

  这是因为 `lc_numeric` 可以潜在的改变这些函数的输出。

- 改善索引对带括号的正则表达式的使用 (Tom)

  这也改善了psql `\d` 的性能。。

E.169. 版本 7.4.14

> 发布日期: 2006-10-16

这个版本包含各种自7.4.13以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.169.1. 迁移到版本 7.4.14

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.169.2. 修改列表

- 修复无类型的文字被当做是ANYARRAY时的内核转储
- 修复 `string_to_array()` 处理分隔符字符串的重叠的匹配

  例如， `string_to_array('123xx456xxx789', 'xx')` 。
- 修复psql的 `\d` 命令中模式匹配的极端情况
- 修复/contrib/ltree中的索引损坏bug (Teodor)
- 修复/contrib/dbmirror中的反斜杠逃逸
- 为最近US DST规律中的改变调整回归测试

E.170. 版本 7.4.13

> 发布日期: 2006-05-23

这个版本包含各种自7.4.12以来的修复，包括非常严重的安全问题的补丁。 关于7.4主版本的新特性的信息，请查阅Section E.183。

E.170.1. 迁移到版本 7.4.13

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

针对CVE-2006-2313和CVE-2006-2314中描述的SQL注入攻击的完全安全可能需要在应用代码中改变。 如果你的应用嵌入了不能信赖的字符串到SQL命令中，你应该尽快检查他们， 以确保他们使用推荐的逃逸技术。在大多数情况下，应用应该使用库或驱动提供的子程序 （如libpq的 `PQescapeStringConn()` ）来执行字符串逃逸， 而不是依赖于*ad hoc*代码实现。

E.170.2. 修改列表

- 改变服务器以在任何情况下都拒绝无效编码的多字节字符 (Tatsuo, Tom)

  PostgreSQL已经朝这个方向发展有一段时间了， 检查现在一致的应用于所有的编码和所有的文本输入中，并且现在总是错误而不仅仅是警告。 这个改变防卫了CVE-2006-2313中描述的SQL注入类型的攻击。

- 拒绝在字符串字面值中不安全的使用 `\'`

  作为服务器端防卫CVE-2006-2314中描述的类型的SQL注入攻击， 该服务器现在在SQL字符串字面值中只接受 `''` ，不接受 `\'` 作为ASCII单引号的表示。缺省的， `\'` 只在 `client_encoding` 设置为仅客户端的编码(SJIS, BIG5, GBK, GB18030, 或 UHC)时拒绝， 这是可能有SQL注入的情况。可用一个新的配置参数 `backslash_quote` 在需要时调节这个行为。请注意，针对CVE-2006-2314的完全安全可能要求客户端侧的改变； `backslash_quote` 的目的是在一定程度上使得不安全的客户端变得明显。

- 修改libpq的字符串逃逸例程， 知道编码注意事项和 `standard_conforming_strings`

  这修复了使用libpq的应用针对CVE-2006-2313和CVE-2006-2314中描述的安全问题， 也预防了转换到SQL标准字符串文字语法的计划。使用多个PostgreSQL 连接并发的应用应该迁移到 `PQescapeStringConn()` 和 `PQescapeByteaConn()` ，以确保每个数据库连接中使用的逃逸设置正确的动作了。 "手动"实现字符串逃逸的应用应该修改为依赖于库例程。

- 修复一些不正确的编码转换函数

  `win1251_to_iso`、`alt_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw`在转换内容时都会损坏。

- 清理`\'`在字符串中剩余的使用 (Bruce, Jan)
- 修复有时会导致where OR条件索引扫描漏掉应该返回的行的bug
- 修复btree索引被截断情况下的WAL重放
- 为包含`|`的模式修复`SIMILAR TO` (Tom)
- 修复服务器正确的使用自定义DH SSL参数 (Michael Fuhr)
- 修复Intel Macs上的Bonjour (Ashley Clark)
- 修复各种小的内存泄露

E.171. 版本 7.4.12

> 发布日期: 2006-02-14

这个版本包含各种自7.4.11以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.171.1. 迁移到版本 7.4.12

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.11的版本升级而来， 请查看7.4.11的版本声明。

E.171.2. 修改列表

- 修复 `SET SESSION AUTHORIZATION` 中潜在的崩溃 (CVE-2006-0553)

  一个非特权的用户可能会使服务器进程崩溃，导致短暂的拒绝对其他用户的服务， 如果服务器编译启用了断言（缺省不启用）。感谢Akio Ishida报告这个问题。

- 修复自动插入的行的行可见性逻辑bug (Tom)

  在极少数情况下，当前命令插入的行可能被视为早就有效了，而此时不应该这样。 修复了在7.4.9和7.3.11版本中创建的bug。

- 修复在pg_clog文件创建期间可能导致"文件早已存在"错误的竞态条件 (Tom)
- 在预备语句中为 `UNKNOWN` 参数适当的检查 `DOMAIN` 约束 (Neil)
- 修复以允许恢复的转储有到自定义操作的交叉模式引用 (Tom)
- 修复配置期间的可移植性，以测试 `finite` 和 `isinf` 的存在 (Tom)

E.172. 版本 7.4.11

> 发布日期: 2006-01-09

这个版本包含各种自7.4.10以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.172.1. 迁移到版本 7.4.11

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.8的版本升级而来， 请查看7.4.8的版本声明。 还有，你可能需要在升级之后 `REINDEX` 在文本字段上的索引， 如果你受下面描述的本地或plperl问题的影响。

E.172.2. 修改列表

- 修复在一个事务外部或一个失败的事务内部发出的协议级别描述信息 (Tom)
- 修复字符串比较认为不同的字符组合是相等的环境，如Hungarian (Tom)

  这可能需要 `REINDEX` 以修复在文本字段上已经有了的索引。
- 在主进程启动期间设置本地环境变量，以确保plperl 不会在稍后改变本地环境

  这修复了一个问题，该问题在postmaster以与initdb 给出的不同的环境变量启动时发生。 在这种情况下，任何plperl 的使用都可能导致索引损坏。如果发生了这种情况，你可能需要 `REINDEX` 以修复在文本字段上已经存在的索引。
- 修复strpos()和正则表达式处理某些很少使用的Asian多字节字符设置中长期存在的bug (Tatsuo)
- 修复 `/contrib/pgcrypto` gen_salt中的bug， 它导致不能使用所有的MD5和XDES算法的盐空间 (Marko Kreen, Solar Designer)

  盐对Blowfish和标准的DES是没有影响的。
- 修复 `/contrib/dblink` 当指定的字段数量与实际查询返回的不同时， 抛出一个错误，而不是崩溃 (Joe)

E.173. 版本 7.4.10

> 发布日期: 2005-12-12

这个版本包含各种自7.4.9以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.173.1. 迁移到版本 7.4.10

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.8的版本升级而来， 请查看7.4.8的版本声明。

E.173.2. 修改列表

- 修复事务日志管理中的竞态条件

  有一个窄的窗口，在这里错误的页面可以发起一个I/O操作，导致维护失败或数据损坏。

- 当当前事务已经中止，如果客户端发送捆绑的协议信息时，阻止失败
- `/contrib/ltree` 修复 (Teodor)
- AIX 和 HPUX 编译修复 (Tom)
- 修复长期存在的外连接的规划错误

  这个bug有时会导致一个假的错误"RIGHT JOIN is only supported with merge-joinable join conditions"。

- 当一个表已经被删除时，阻止pg_autovacuum中的内核转储

E.174. 版本 7.4.9

> 发布日期: 2005-10-04

这个版本包含各种自7.4.8以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.174.1. 迁移到版本 7.4.9

运行7.4.X的用户不需要转储/恢复。不过，如果你是从一个早于7.4.8的版本升级而来， 请查看7.4.8的版本声明。

E.174.2. 修改列表

- 修复允许 `VACUUM` 移除 `ctid` 链太快的错误， 并且在跟随 `ctid` 连接的代码中添加更多检查

  这修复了一个可能在非常少的情况下导致崩溃的长期存在的问题。

- 修复当使用一个多字节字符设置时， `CHAR()` 适当的填补空格到指定的长度 (Yoshiyuki Asaba)

  在以前的版本中， `CHAR()` 的填补是不正确的，因为它只填补到指定的字节数量， 而不考虑存储多少个字节。

- 修复 `COPY` 只读事务测试的感觉

  代码以前禁止 `COPY TO` ， 实际上应该禁止的是 `COPY FROM` 。

- 修复子句上的外连接只引用内侧关系的规划问题
- 进一步修复 `x FULL JOIN y ON true` 的极端情况
- 使得 `array_in` 和 `array_recv` 更执着的验证它们的OID参数
- 修复像 `UPDATE a=... WHERE a...` with GiST index on column `a` 这样的查询中丢失的行
- 提高日期时间解析的鲁棒性
- 改善检查部分写入的WAL页面
- 改善当启用了SSL时信号处理的鲁棒性
- 在主进程启动期间不要尝试打开超过 `max_files_per_process` 的文件

- 修复各种内存泄露
- 提高各种可移植性
- 修复当变量是引用传递类型时，PL/pgSQL正确的处理`var := var`
- 更新`contrib/tsearch2`以使用现在的Snowball代码

E.175. 版本 7.4.8

> 发布日期: 2005-05-09

这个版本包含各种自7.4.7以来的修复，包括几个安全相关的问题。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.175.1. 迁移到版本 7.4.8

运行7.4.X的用户不需要转储/恢复。不过， 有可能在7.4.X系统日志最初的内容中处理两个已经发现的重大安全问题。 一个使用7.4.8的dump/initdb/reload序列initdb将自动纠正这些问题。

较大的安全问题是内建字符集编码转换函数可以被非特权的用户从SQL命令调用， 但是该函数不是设计来这样使用的，并且在恶意的选择参数时是不安全的。 该修复包括改变这些函数声明的参数列表，这样他们可以不再从SQL命令调用。 （这不影响他们通过编码转换机制的正常使用。）

较小的问题是 `contrib/tsearch2` 模块创建了几个函数， 这几个函数错误的声明为返回 `internal` 而它们不接受 `internal` 参数。 这破坏了所有使用 `internal` 参数的函数的类型安全。

强烈建议所有的安装都修复这些错误，通过initdb或者通过下面给出的手动修复程序。 该错误至少允许未授权的数据库用户崩溃他们的服务器进程， 并且可能允许未授权的用户获取数据库超级用户的权限。

如果不想做initdb，那么执行下面的程序。作为数据库超级用户，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
     AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

下一步，如果你已经安装了 `contrib/tsearch2` ，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
   'dex_init(text)'::regprocedure,
   'snb_en_init(text)'::regprocedure,
   'snb_ru_init(text)'::regprocedure,
   'spell_init(text)'::regprocedure,
   'syn_init(text)'::regprocedure
);
-- The command should report having updated 5 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

如果这个命令带有像"function "dex_init(text)" does not exist" 这样的消息失败，那么要么在这个数据库中没有安装 `tsearch2`， 要么你已经执行了该更新。

上面的程序必须在每个安装的数据库中执行，包括 `template1`， 理想上也包括 `template0`。如果你没有修复模板数据库， 那么任何随后创建的数据库都将包含相同的错误。 `template1` 的修复方式和其他数据库相同， 但是修复 `template0` 需要额外的步骤。首先，从任意数据库中发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

下一步，连接到 `template0`，并执行上面的修复步骤。最后，执行：

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.175.2. 修改列表

- 改变编码函数签名以阻止滥用
- 修改 `contrib/tsearch2` 以避免不安全的使用 `INTERNAL` 函数的结果
- 修复允许一个事务因为某些原因（如SELECT FOR UPDATE）比其他原因稍早的被看做是已提交的过时的竞态条件

  这是一个非常严重的bug，因为它会导致表面上的数据不一致被应用短暂的看到。

- 修复关系扩展和VACUUM之间的竞态条件

  这理论上会导致最近插入的一页数据丢失，尽管这种情况看起来可能性非常小。 没有已知的情况说它会引起超过一个维护的失败。

- 修复 `TIME WITH TIME ZONE` 值的比较

当使用了`--enable-integer-datetimes`配置开关时，比较代码是错误的。 注意：如果你在`TIME WITH TIME ZONE`字段上有一个索引， 它将需要在安装这个更新之后`REINDEX`，因为该修复纠正了字段值的排序顺序。

- 为`TIME WITH TIME ZONE`值修复`EXTRACT(EPOCH)`
- 修复负的分数秒在`INTERVAL`值中的错误显示

  这个错误只在使用了`--enable-integer-datetimes`配置开关的时候发生。

- 确保在后端关闭期间所做的操作都被统计收集器计数了

  这预计能解决pg_autovacuum清理系统目录不够频繁的报告— 没有被告知在后端退出期间由于临时表删除引起的目录删除。

- 在plpgsql中追加缓冲区溢出检查 (Neil)
- 修复pg_dump以正确的转储名字包含`%`的触发器 (Neil)
- 为更新的OpenSSL建立修复`contrib/pgcrypto` (Marko Kreen)
- 为`contrib/intagg`更多的64位修复
- 阻止返回`RECORD`的函数不正确的最优化
- 阻止`to_char(interval)`转储月相关格式的内核
- 阻止`COALESCE(NULL,NULL)`上的崩溃
- 修复`array_map`以正确的调用PL函数
- 修复`ALTER DATABASE RENAME`中的权限检查
- 修复`ALTER LANGUAGE RENAME`
- 使得`RemoveFromWaitQueue`清理它本身

  这修复了一个锁管理错误，该错误只在这种情况下可以看到： 如果一个事务被从锁等待中踢出（通常通过查询取消）， 并且然后锁的持有者在一个非常小的窗口释放它。

- 修复无类型的参数出现在`INSERT ... SELECT`中的问题
- 修复`ALTER TABLE SET WITHOUT OIDS`之后的`CLUSTER`失败

E.176. 版本 7.4.7

> 发布日期: 2005-01-31

这个版本包含各种自7.4.6以来的修复，包括几个安全相关的问题。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.176.1. 迁移到版本 7.4.7

运行7.4.X的用户不需要转储/恢复。

E.176.2. 修改列表

- 不允许非超级用户 `LOAD`

  在自动执行一个共享库的初始化函数的平台上（至少包括Windows和ELF-based Unixen）， `LOAD` 可以用来使服务器执行任意的代码。感谢NGS Software报告这个问题。

- 检查聚集函数的创建者是否有权限执行指定的转换函数

  这个疏忽使得它有可能绕开一个函数的EXECUTE权限的拒绝。

- 在contrib/intagg中修复安全和64位问题
- 添加需要的STRICT标记到一些贡献函数 (Kris Jurka)
- 当plpgsql游标声明有太多的参数时，避免缓冲区溢出 (Neil)
- 修复FULL和RIGHT外连接的规划错误

  连接的结果错误的假设为和左侧的输入排序相同。这不只会交付错误排序的输出给用户， 还会在嵌套合并连接的情况下给出完全错误的答案。

- 修复plperl在元组字段中的引用标记
- 修复负的间隔在SQL和GERMAN日期类型中的显示
- 使age(timestamptz)用本地时区计算，不是用GMT计算

E.177. 版本 7.4.6

> 发布日期: 2004-10-22

这个版本包含各种自7.4.5以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.177.1. 迁移到版本 7.4.6

运行7.4.X的用户不需要转储/恢复。

E.177.2. 修改列表

- 修复在磁盘上更新提示部分可能的失败

  在少数情况下，这个疏忽会导致"could not access transaction status"失败， 这使得它成为一个潜在的数据丢失bug。

- 确保散列的外连接不丢失元组

  使用散列连接规划的非常大的左连接可能不能输出不匹配的左侧行， 只给出右侧数据分布。

- 不允许作为root运行pg_ctl

  这是为了防卫任何可能的安全问题。

- 避免在 `/tmp` 中以 `make_oidjoins_check` 使用临时文件

  这已经报告为一个安全问题，尽管它几乎不值得考虑，因为非开发者没有理由使用这个脚本。

- 阻止强制的后端关闭重新发出以前的命令结果

  在稀有情况下，一个客户端可能认为它的最后一个命令已经成功， 而实际上它已经由于强制数据库关闭而退出了。

- 修复 `pg_stat_get_backend_idset` 中的bug

  这会导致一些系统统计视图中的错误行为。

- 修复主进程中小的内存泄露

- 修复"expected both swapped tables to have TOAST tables"错误

这会在例如在ALTER TABLE DROP COLUMN之后CLUSTER的情况下出现。

- 阻止 `pg_ctl restart` 添加 `-D` 多次
- 修复NULL值在GiST索引中的问题
- `::` 不再看做是一个ECPG预备语句中的变量

E.178. 版本 7.4.3

> 发布日期: 2004-08-18

这个版本包含一系列对7.4.4的错误修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.178.1. 迁移到版本 7.4.5

运行7.4.X的用户不需要转储/恢复。

E.178.2. 修改列表

- 修复并发的B-tree索引插入期间可能的崩溃

  这个补丁修复了并发插入到B-tree索引可能导致服务器恐慌的稀有情况。 不会导致永久的损坏，但是仍然需要重新发布。这个错误在7.4以前的版本中不存在。

E.179. 版本 7.4.4

> 发布日期: 2004-08-16

这个版本包含各种自7.4.3以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.179.1. 迁移到版本 7.4.4

运行7.4.X的用户不需要转储/恢复。

E.179.2. 修改列表

- 阻止崩溃期间丢失已提交事务的可能

  由于事务提交和检查点之间不充分的连锁，有可能事务提交刚好在最近的一个检查点丢失之前， 整体的或部分的，跟随着一个数据库崩溃并且重启。这是自PostgreSQL 7.1 就已经存在的一个严重的错误。

- 在评估聚合规划的结果列表之前检查HAVING约束
- 避免会话的当前用户ID被删除时的崩溃
- 修复零行情况下的散列交叉表 (Joe)
- 强制在一个外键中重命名一个字段之后更新缓存
- 正确的打印UNION查询
- 使得psql在COPY IN中适当的处理 `\r\n` 新行
- pg_dump错误的使用授予的选项处理ACL
- 修复对OS X和Solaris的线性支持
- 使用各种修复更新JDBC驱动(build 215)
- 修复ECPG
- 更新翻译（各种各样的贡献者）

E.180. 版本 7.4.3

> 发布日期: 2004-06-14

这个版本包含各种自7.4.2以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.180.1. 迁移到版本 7.4.3

运行7.4.X的用户不需要转储/恢复。

E.180.2. 修改列表

- 修复使用非散列的聚集时的临时内存泄露 (Tom)
- 修复ECPG，包括一些Informix兼容性 (Michael)
- 修复线程安全的编译，尤其是Solaris (Bruce)
- 修复使用老的网络协议时COPY IN终止中的错误 (ljb)
- 几个pg_autovacuum的重要的修复，包括修复大表、无符号的oid、稳定性、 临时表和调试模式 (Matthew T. O'Connor)
- 修复在NetBSD和BSD/OS上读取tar格式转储的问题 (Bruce)
- 几个JDBC修复
- 修复last_value等于重新开始值时的ALTER SEQUENCE RESTART (Tom)
- 修复未能重新计算嵌套的子查询 (Tom)
- 修复LIMIT/OFFSET中的非常数表达式的问题
- 支持没有连接子句的FULL JOIN，比如X FULL JOIN Y ON TRUE (Tom)
- 修复另一个零字段表错误 (Tom)
- 改善子查询的GROUP BY子句中非限定标识符的处理 (Tom)

  子查询中选择列表的别名的优先级现在高于外查询级别的名字。
- 解码规则时不要生成"NATURAL CROSS JOIN" (Tom)
- 添加对二进制COPY中无效字段长度的检查 (Tom)

这修复了难以利用的安全漏洞。

- 避免`ANALYZE`和`LISTEN`/`NOTIFY`之间的锁冲突
- 许多翻译更新 (各种各样的贡献者)

E.181. 版本 7.4.2

> 发布日期: 2004-03-08

这个版本包含各种自7.4.1以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.181.1. 迁移到版本 7.4.2

运行7.4.X的用户不需要转储/恢复。不过， 作为合并修复在最初的7.4.X系统目录中发现的两个错误的最简单的方法，这应该会是明智的。 使用7.4.2的initdb的dump/initdb/reload序列将自动纠正这些问题。

更严重的两个错误是数据类型 `anyarray` 有错误的对齐标签； 这是一个问题，因为 `pg_statistic` 系统目录使用 `anyarray` 字段。 贴错标签会引起规划器错误估计甚至当包含 `WHERE` 子句的规划器查询在两端对齐的字段(如 `float8` 和 `timestamp`)上时会崩溃。 强烈建议所有安装都修复这个错误，通过initdb或遵循下面给出的手动修复程序。

较小的错误是系统视图 `pg_settings` 应该被标记为有公共更新访问， 以允许 `UPDATE pg_settings` 用作 `SET` 的替代。 这也可以通过initdb或者手动修复，但是没有必要修复，除非你想要使用 `UPDATE pg_settings` 。

如果你不愿意做initdb，那么下面的程序将修复 `pg_statistic` 。 作为数据库超级用户，执行：

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

这可以在活动的数据库中完成，但是要注意所有运行在改变了的数据库中的后端都必须在 重新填充 `pg_statistic` 是安全的之前重新启动。

要修复 `pg_statistic` 错误，只需要做：

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

上面的程序必须在每个安装的数据库中执行，包括 `template1` ， 理想上也包括 `template0` 。如果你没有修复模板数据库， 那么任何随后创建的数据库都将包含相同的错误。 `template1` 的修复方式和其他数据库相同， 但是修复 `template0` 需要额外的步骤。首先，从任意数据库中发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

下一步，连接到 `template0` ，并执行上面的修复步骤。最后，执行：

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.181.2. 修改列表

版本7.4.2合并了所有包含在版本7.3.6中的修复，加上下面的修复：

- 修复会使优化器崩溃的 `pg_statistics` 对齐错误

  关于这个问题的详细信息请查看上文。

- 允许非超级用户更新 `pg_settings`
- 修复几个优化器错误，大多数会导致"variable not found in subplan target lists"错误
- 避免启动大的多个索引扫描期间的超出内存失败
- 修复 `COPY IN` 期间可能导致"内存不足"错误的多字节问题
- 修复没有OID的表的 `SELECT INTO` / `CREATE TABLE AS` 问题
- 修复并行测试期间 `alter_table` 回归测试的问题
- 修复打开文件限制的问题，尤其是在OS X上 (Tom)
- 部分的修复Turkish区域设置问题

  initdb在Turkish的区域设置现在将成功，但是仍然有一些与 `i/I` 问题相关的不便。

- 使得pg_dump在恢复时设置客户端编码
- 其他较小的pg_dump修复
- 允许ecpg再次使用C关键字作为字段名 (Michael)
- 添加ecpg `WHENEVER NOT_FOUND` 到 `SELECT/INSERT/UPDATE/DELETE` (Michael)
- 修复查询调用设置返回函数时的ecpg崩溃 (Michael)

- 各种其他ecpg修复 (Michael)
- 修复Borland编译器
- 线程构建的改进 (Bruce)
- 各种其他构建修复
- 各种JDBC修复

E.182. 版本 7.4.1

> 发布日期: 2003-12-22

这个版本包含各种自7.4以来的修复。关于7.4主版本的新特性的信息， 请查阅Section E.183。

E.182.1. 迁移到版本 7.4.1

运行7.4的用户不需要转储恢复。

如果你想在信息模式中安装该修复，你需要重新加载它到数据库。 通过运行 `initdb` 初始化一个新的集群， 或者通过在每个数据库中（理想上包括 `template1` ） 作为超级用户在psql中运行下面的SQL命令序列来完成， 安装新的版本之后：

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

用第二个命令替换你的安装路径。

E.182.2. 修改列表

- 用ECPG修复 `CREATE SCHEMA` 解析器中的bug (Michael)
- 修复 `--enable-thread-safety` 和 `--with-perl` 一起使用时的编译错误 (Peter)
- 修复使用哈希连接的子查询 (Tom)

  使用哈希连接的某些子查询将会由于不适当的共享结构崩溃。
- 修复自由空间映射压缩错误 (Tom)

  这修复了压缩自由空间映射会导致数据库服务器关闭的错误。
- 修复用libpq构造的Borland编译器 (Bruce)
- 修复 `netmask()` 和 `hostmask()` 以返回最大长度的masklen （Tom）

  修复这些函数以返回和7.4以前的版本一致的值。
- 几个 `contrib/pg_autovacuum` 修复

  修复包括不正确的变量初始化，在 `TRUNCATE` 之后忘记清理， 和持续计算长的清理的溢出。

- 允许在Cygwin下编译 `contrib/cube` (Jason Tishler)
- 当没有定义口令时，允许Solaris使用口令文件 (Tom)

  修复没有定义口令时，由于使用任意类型的口令认证引起的Solaris崩溃。
- JDBC修复线程问题，和其他的修复
- 修复 `bytea` 索引查找 (Joe)
- 为位数据类型修复信息模式 (Peter)
- 在从WAL恢复期间强制zero_damaged_pages为on
- 阻止一些"变量不在子计划目标列表中"的模糊情况
- 使 `PQescapeBytea` 和 `byteaout` 相互之间一致 (Joe)
- 为字节bytes > 0x7e逃逸 `bytea` 输出 (Joe)

  如果不同的客户端编码用于 `bytea` 输出和输入， 那么 `bytea` 值有可能会被不同的编码损坏。这个修复逃逸了所有可能被影响的字节。
- 添加丢失的 `SPI_finish()` 调用到dblink的 `get_tuple_of_interest()` (Joe)
- 新增Czech FAQ
- 为外键修复信息模式视图 `constraint_column_usage` (Peter)
- ECPG修复 (Michael)
- 修复子查询中的多个 `IN` 子查询和连接的错误 (Tom)
- 允许 `COUNT('x')` 工作 (Tom)
- 为Informix兼容性安装ECPG包含文件到单独的目录 (Peter)

  一些Informix兼容性ECPG包含文件的名字与操作系统包含文件冲突。 通过将他们安装到他们自己的目录中，名字冲突就减少了。
- 修复SSL内存泄露 (Neil)

  这个版本修复了7.4中SSL不释放任何它分配到的内存的错误。
- 阻止 `pg_service.conf` 使用服务名作为缺省的dbname (Bruce)
- 修复FreeBSD上的本地身份认证 (Tom)

E.183. 版本 7.4

> 发布日期: 2003-11-17

E.183.1. 概述

这个版本中的主要变更：

`IN` / `NOT IN` 子查询现在更有效率

在以前的版本中， `IN` / `NOT IN` 子查询连接到上面的查询，顺序的扫描子查询需找一个匹配。 7.4代码使用普通连接使用的相同的先进的技术，因此更加快速。 `IN` 现在通常比一个等价的 `EXISTS` 子查询相同速度或者更快； 这颠倒了应用于以前版本的传统观点。

通过使用哈希桶提高 `GROUP BY` 的处理

在以前的版本中，被分组的行必须先排序。7.4代码可以不用排序就执行 `GROUP BY` ， 通过累加结果到一个哈希表，每个分组都有一个入口。不过， 如果该哈希表被估计为太大而不适合 `sort_mem` ，那么将仍然使用排序技术。

新增多键的哈希连接能力

在以前的版本中，哈希连接只在单键上发生。这个版本允许多键的哈希连接。

使用明确的 `JOIN` 语法的查询现在更好的最优化了

以前的版本估算使用明确的 `JOIN` 语法的查询只以该语法暗示的顺序。 7.4允许全面优化这些查询，意味着优化考虑所有可能的连接顺序， 并且选择最有效的顺序。不过，外连接必须仍然遵循声明的顺序。

更快、更强大的正则表达式编码

整个正则表达式模块已经被Henry Spencer用一个新的版本替代，最初是为Tcl写的。 该代码大大的提高了性能，并且支持几种正则表达式。

为简单的SQL函数嵌入函数

简单的SQL函数现在可以通过在主查询中包含他们的SQL内联。 这通过消除每个调用的开销提升了性能。这意味着简单的SQL函数现在的行为就像宏。

全面支持IPv6连接和IPv6地址数据类型

以前的版本只允许IPv4连接，IP数据类型也只支持IPv4地址。 这个版本在这些区域添加了完全的IPv6支持。

主要改进SSL性能和可靠性

几个人非常了解SSL API已经彻底检查了我们的SSL代码以改善SSL键谈判和错误恢复。

使得自由空间映射有效的再利用空的索引页，其他自由空间管理的改善

在以前的版本中，因为删除的行只能被索引值类似于该行以前在该页上的索引的行再次利用， 所以B-tree索引页都保留空白。在7.4中， `VACUUM` 记录空的索引页，并且允许它们被任意未来的索引行重新利用。

SQL标准信息模式

该信息模式提供一个标准且稳定的方式访问关于在一个数据库中定义的模式对象的信息。

游标更接近的符合SQL标准

命令 `FETCH` 和 `MOVE` 已经被彻底检查，更加符合SQL标准。

游标可以在事务之外退出

这些游标也叫做可抓住的游标。

新增客户端到服务器的协议

新的协议添加了错误代码，更多的状态信息，更快的启动，更好的支持二进制数据传输， 参数值从SQL命令中分离，预备语句在协议级别中有效，更干净的从 `COPY` 失败中恢复。服务器和客户端仍然支持老的协议。

libpq和ECPG 应用现在是完全线程安全的

以前的libpq版本就已经支持了线程， 这个版本通过修复一些在数据库连接启动期间使用的非线程安全的代码提高了线程安全。 `configure` 选项 `--enable-thread-safety` 必须用于启用这个特性。

新增全文本索引的版本

一个新的全文本索引套件在 `contrib/tsearch2` 中可用。

新增自动清理工具

`contrib/autovacuum` 中新增的自动清理工具见识数据库状态表的 `INSERT` / `UPDATE` / `DELETE` 活动并在需要时自动清理表。

改进了数组处理并嵌入服务器内核

移除了许多数组限制，并且数组表现的更加全面支持数据类型。

E.183.2. 迁移到版本 7.4

想要从以前的版本迁移数据的用户需要使用pg_dump转储/恢复。

观察下面的不兼容性：

- 删除了服务器端的自动提交设置，并且在客户端应用和语言中重新实现。 服务器端的自动提交导致许多想要控制他们自己的自动提交行为的语言和应用出现问题， 所以从服务器删除自动提交并适当的添加到单独的客户端API中。
- 错误消息的表达在这个版本中已经大幅度的改变了。花费了巨大的努力使得消息更加一致和面向用户。 如果您的应用程序尝试通过分析错误消息检测不同的错误条件， 强烈建议您使用新的错误代码功能。
- 使用明确的 `JOIN` 语法的内连接可能表现的不一致， 因为它们现在更好的优化了。
- 一些服务器配置参数已经重命名的更加清晰了，主要是那些与记录相关的参数。
- `FETCH 0` 或 `MOVE 0` 现在什么也不做。 在以前的版本中， `FETCH 0` 将抓取所有剩余的行， `MOVE 0` 将移动到游标的最后。
- `FETCH` 和 `MOVE` 现在返回实际抓取/移动的行数， 或者如果在游标的开始或结尾返回零。 以前的版本将返回传递给命令的行计数， 不是实际抓取或移动的行数。
- `COPY` 现在可以处理使用回车键或回车键/换行符行尾序列的文件。 字面的回车键和换行不再被数据值接受；使用 `\r` 和 `\n` 替代。
- 当从类型 `char(``_n_` )到 `varchar(``_n_` ) 或 `text` 转换时，现在会去除结尾的空白。这是大多数人们一直想要实现的。
- 数据类型 `float(``_p_` )现在以二进制位数测量 `_p_` ， 不是十进制位数。新的行为遵从SQL标准。
- 模糊的日期值现在必须匹配 `datestyle` 设置指定的顺序。 在以前的版本中，一个 `10/20/03` 的日期声明理解为十月中的一个日期， 即使 `datestyle` 声明了天在前面。如果对于当前的 `datestyle` 设置来说，日期声明是无效的时7.4将抛出一个错误。
- 函数 `oidrand` 、 `oidsrand` 和 `userfntest` 已经移除了。这些函数不再有用了。
- 字符串字面值声明随时变化的日期/时间值，如 `'now'` 或 `'today'` 在字段缺省表达式中将不再像预期的那样工作； 它们现在缺省是表创建的时间，而不是插入的时间。 应该使用像 `now()` 、 `current_timestamp` 或 `current_date` 这样的函数。

  在以前的版本中，有特殊的代码，所以像 `'now'` 这样的代码被解释为 `INSERT` 时间，而不是表创建的时间， 但是这个变通并不涵盖所有的情况。版本7.4现在要求该缺省使用像 `now()` 或 `current_timestamp` 这样的函数正确的定义。 这将适用于所有的情况。

- 美元符号(`$`)不再允许用于操作员名字。它反而可以成为标识符中的非首位字符。 这样做是为了提高和其他数据库系统的兼容性，和避免参数占位符 (`$``_n_`)和操作符写的很近时的语法问题。

E.183.3. 修改列表

在下面你将发现版本7.4和前面的主要版本之间的详细的变化。

E.183.3.1. 服务器操作变化

- 允许IPv6服务器连接(Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- 修复SSL以干净利落的处理错误 (Nathan Mueller)

  在以前的版本中，某些SSL API错误报告处理的不正确。这个版本修复了那些问题。

- SSL协议安全和性能改善 (Sean Chittenden)

  SSL键重新谈判非常频繁的发生， 导致SSL性能低下。 另外，改善了最初的键的处理。

- 当检测到死锁时打印锁信息 (Tom)

  这允许更容易的调试死锁状况。

- 更新 `/tmp` 套接字修改时间，定期的避免他们的移除 (Tom)

  这应该有助于防止 `/tmp` 目录清理管理脚本移除服务器套接字文件。

- 为Mac OS X启用PAM (Aaron Hillegass)
- 使得B-tree索引完全的WAL安全 (Tom)

  在以前的版本中，在某些稀有情况下，服务器崩溃会导致B-tree索引损坏。 这个版本删除了最后的几个稀有情况。

- 允许B-tree索引压缩和空页面重新利用 (Tom)
- 在第一个根页面分裂期间修复不一致的索引查找 (Tom)

  在以前的版本中，当一个页面索引分裂为两个页面时， 另一个数据库会话会在短期内看不到索引项。 这个版本修复了这个稀有的失败情况。

- 改善自由空间映射分配逻辑 (Tom)
- 在服务器重启时保留自由空间信息 (Tom)

在以前的版本中，当主进程停止时，自由空间映射是不保存的， 所以新启动的服务器没有自由空间信息。这个版本保存了自由空间映射， 并且当服务器重新启动时重新加载它。

- 添加开始时间到 `pg_stat_activity` (Neil)
- 新增代码检测损坏的磁盘页；删除 `zero_damaged_pages` (Tom)
- 新增客户端/服务器协议：更快、没有用户名长度限制、允许从 `COPY` 中干净的退出 (Tom)
- 添加事务状态、表ID、字段ID到客户端/服务器协议 (Tom)
- 添加二进制I/O到客户端/服务器协议 (Tom)
- 删除自动提交服务器设置；移动到客户端应用 (Tom)
- 新的错误消息用词、错误代码和三个层次的错误详情 (Tom, Joe, Peter)

E.183.3.2. 性能改进

- 为 `GROUP BY` 聚集添加散列 (Tom)
- 使得嵌套循环连接关于多字段索引更加智能 (Tom)
- 允许多键的哈希连接 (Tom)
- 提高常量折叠 (Tom)
- 添加内联简单的SQL函数的能力 (Tom)
- 减少使用复杂函数的查询的内存使用 (Tom)

  在以前的版本中，返回存储器配置的函数将直到查询完成之后才释放。 这个版本允许调用完成之后就释放存储器配置函数，减少函数使用的总内存。

- 提高GEQO优化器性能 (Tom)

  这个版本修复了几个GEQO优化器管理潜在查询路径的低效率。

- 允许通过哈希表处理 `IN` / `NOT IN` (Tom)
- 提高 `NOT IN (``_subquery_`)性能 (Tom)
- 允许大多数 `IN` 子查询作为连接处理 (Tom)
- 模式匹配操作可以使用索引，无视区域设置 (Peter)

  非ASCII区域设置无法为 `LIKE` 比较使用标准的索引。 这个版本添加了为 `LIKE` 创建特殊索引的方法。

- 允许主进程使用 `preload_libraries` 预加载库 (Joe)

  对于需要很长时间加载的共享库，这个选项是可用的，这样库可以在主进程中预加载并被所有的数据库会话继承。

- 改善优化器成本计算，尤其是子查询 (Tom)
- 避免子查询 `ORDER BY` 匹配上一级查询时的排序 (Tom)
- 推断 `WHERE a.x = b.y AND b.y = 42` 的意思也为 `a.x = 42` (Tom)
- 允许在复杂连接上哈希/合并连接 (Tom)
- 允许为更多数据类型哈希连接 (Tom)
- 允许明确的内连接最优化，禁用 `join_collapse_limit` (Tom)
- 添加参数 `from_collapse_limit` 控制子查询到连接的转换 (Tom)
- 使用更快更强大的Tcl正则表达式代码 (Henry Spencer, Tom)
- 在优化器中使用位映射关系设置 (Tom)
- 改善连接启动时间 (Tom)

  新的客户端/服务器协议需要较少的网络数据包开始一个数据库会话。

- 提高触发器/约束性能 (Stephan)
- 提高 `col IN (const, const, const, ...)` 的速度 (Tom)
- 修复在稀有情况下损坏的哈希索引 (Tom)
- 改善哈希索引并发性和速度 (Tom)

  以前版本的哈希索引性能欠佳，尤其是高并发性的情况下。这个版本修复了这个问题，并且开发组对于报告B-tree和哈希索引性能的比较是有兴趣的。

- 在32字节的边界上排列共享的缓冲区，改善拷贝速度 (Manfred Spraul)

  当地址是32字节对齐的时，确定的CPU执行数据拷贝更快。

- 重新使用数据类型 `numeric`，以获取更好的性能 (Tom)

  `numeric` 过去是基于100存储。新的代码基于10000，大大的改善性能。

E.183.3.3. 服务器配置变化

- 重命名服务器参数 `server_min_messages` 为 `log_min_messages` (Bruce)

  这样做了之后大部分控制服务器日志的参数以 `log_` 开始。

- 重命名 `show_*_stats` 为 `log_*_stats` (Bruce)
- 重命名 `show_source_port` 为 `log_source_port` (Bruce)
- 重命名 `hostname_lookup` 为 `log_hostname` (Bruce)
- 添加 `checkpoint_warning` 警告过多的检查点 (Bruce)

  在以前的版本中，很难判断检查点是否发生的太过频繁。 这个特性当过多的检查点发生时向服务器日志添加一个警告。

- 为本地化新增只读服务器参数 (Tom)
- 改变调试服务器日志消息使输出 `DEBUG` 而不是 `LOG` (Bruce)
- 阻止服务器日志变量被非超级用户关闭 (Bruce)

  这是一个安全特性，所以非超级用户不能禁用管理员启用的日志。

- `log_min_messages` / `client_min_messages` 现在控制 `debug_*` 输出 (Bruce)

  这集中了客户端调试信息，所以所有的调试输出都可以发送到客户端和服务器日志。

- 添加Mac OS X集合点服务器支持 (Chris Campbell)

  这允许Mac OS X主机为可用的PostgreSQL服务器查询网络。

- 添加使用 `log_min_duration_statement` 只打印缓慢的语句的能力 (Christopher)

  这是一个经常请求的调试特性，允许管理员在他们的服务器日志中只看到缓慢的查询。

- 允许 `pg_hba.conf` 在CIDR格式中接受子网掩码 (Andrew Dunstan)

  这允许管理员在 `pg_hba.conf` 中合并主机IP地址和子网掩码字段为一个CIDR字段。

- 新增只读参数 `is_superuser` (Tom)
- 新增参数 `log_error_verbosity` 控制错误详情 (Tom)

  与新的错误报告特性一起工作，提供额外的错误信息，如提示、文件名和行号。

- `postgres --describe-config` 现在转储服务器配置变量 (Aizaz Ahmed, Peter)

  这个选项对于需要知道配置变量名和它们的最小、最大、缺省和描述的管理工具来说是有用的。

- 在 `pg_settings` 中添加新的字段： `context` , `type` , `source` , `min_val` , `max_val` (Joe)
- 如果可能，使 `shared_buffers` 缺省为1000， `max_connections` 缺省为100 (Tom)

以前的版本的共享缓冲区缺省为64，所以PostgreSQL 甚至在非常老的系统上也能启动。这个版本测试平台允许的共享内存的数量并且如果可以的话选择更加合理的缺省值。 当然，仍然鼓励用户评估他们的资源负载，并且相应的设定 `shared_buffers` 的尺寸。

- 新增 `pg_hba.conf` 记录类型 `hostnossl` 以阻止SSL连接 （Jon Jensen）

  在以前的版本中，如果客户端和服务器都支持SSL，那么是没有办法阻止SSL连接的。这个选项允许阻止SSL连接。

- 删除参数 `geqo_random_seed` (Tom)
- 添加服务器参数 `regex_flavor` ，控制正则表达式处理 (Tom)
- 使 `pg_ctl` 更好的处理非标准的端口 (Greg)

E.183.3.4. 查询的变化

- 新增SQL标准信息模式 (Peter)
- 添加只读事务 (Peter)
- 在外键违反信息中输出键名和值 (Dmitry Tkach)
- 允许用户在 `pg_stat_activity` 中查看他们自己的查询 (Kevin Brown)

  在以前的版本中，只有超级用户可以使用 `pg_stat_activity` 查看查询字符串。 现在普通用户也可以查看他们自己的查询字符串了。

- 修复子查询中的聚集以匹配SQL标准 (Tom)

  SQL标准认为在一个嵌套的子查询中出现的聚集函数，如果它的参数只包含外部查询的变量， 那么它属于外部的查询。以前的PostgreSQL 版本没有正确的处理这点。

- 添加选项阻止查询中引用的表的自动添加 (Nigel J. Andrews)

  缺省的，查询中提到的表如果没有在 `FROM` 子句中则自动添加进去。 这是和历史的POSTGRES行为兼容的，但是违反SQL标准。 这个选项允许选择标准兼容的行为。

- 允许 `UPDATE ... SET col = DEFAULT` (Rod)

  这允许 `UPDATE` 设置一个字段为它声明的缺省值。

- 允许在 `LIMIT` / `OFFSET` 中使用表达式 (Tom)

  在以前的版本中， `LIMIT` / `OFFSET` 只能使用约束，不能使用表达式。

- 实现了 `CREATE TABLE AS EXECUTE` (Neil, Peter)

E.183.3.5. 对象操作的改变

- 使 `CREATE SEQUENCE` 语法更加符合SQL:2003 (Neil)
- 添加语句级别的触发器 (Neil)

  虽然这允许触发器在语句结束时触发，但是它不允许触发器访问语句修改的所有行。 这个能力是为未来的版本计划的。

- 为域添加检查约束 (Rod)

  通过允许域使用检查约束，大大的增加了域的有用性。

- 新增 `ALTER DOMAIN` (Rod)

  这允许操作现有的域。

- 修复几个零字段表错误 (Tom)

  PostgreSQL支持零字段表。这修复了各种使用这样的表时发生的错误。

- 让 `ALTER TABLE ... ADD PRIMARY KEY` 添加非空约束 (Rod)

  在以前的版本中， `ALTER TABLE ... ADD PRIMARY KEY` 将添加一个唯一索引，但是不添加非空约束。这在这个版本中修复了。

- 添加了 `ALTER TABLE ... WITHOUT OIDS` (Rod)

  这允许控制无论新行还是更新的行都将有一个OID字段。这对于节省存储空间非常有用。

- 添加 `ALTER SEQUENCE` 修改最小、最大、增量、缓存、循环值 (Rod)
- 添加了 `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)

  这个命令被 `pg_dump` 使用，为每个以前集群的表记录集群字段。 这个信息被数据库范围的集群使用以集群所有以前集群的表。

- 为域增加自动类型转换 (Rod, Tom)
- 允许美元符号出现在标识符中，除了作为第一个字符 (Tom)
- 不允许美元符号出现在操作符名中，所以 `x=$1` 工作 (Tom)
- 允许使用 `LIKE` `_subtable_` 拷贝表模式， 这也是SQL:2003的特性 `INCLUDING DEFAULTS` (Rod)
- 添加 `WITH GRANT OPTION` 子句到 `GRANT` (Peter)

  这启用了 `GRANT` ，给予其他用户在对象上授予权限的能力。

E.183.3.6. 实用命令的改变

- 为临时表添加 `ON COMMIT` 子句到 `CREATE TABLE` (Gavin)

  这添加了在事务提交时表被删除或删除所有行的能力。

- 允许游标使用 `WITH HOLD` 在事务之外 (Neil)

  在以前的版本中，游标在创建它们的事务的结束删除。 现在游标可以使用 `WITH HOLD` 选项创建， 这允许它们在创建它们的事务提交之后仍然可以被访问。

- `FETCH 0` 和 `MOVE 0` 现在什么也不做 (Bruce)

  在以前的版本中， `FETCH 0` 抓取所有现有的行， `MOVE 0` 移动到游标的最后。

- 使 `FETCH` 和 `MOVE` 返回抓取/移动的行数，或者如果在游标的开始/结束则为零，按照SQL标准 (Bruce)

  在以前的版本中， `FETCH` 和 `MOVE` 返回的行计数不能精确的反映处理的行数。

- 正确的处理带有游标的 `SCROLL` ，或者报告一个错误 (Neil)

  允许到某种查询的随机访问（前向和后向滚动）不能不带有一些额外的工作完成。 如果 `SCROLL` 在游标创建时指定，这个额外的工作将被执行。 此外，如果游标带有 `NO SCROLL` 创建，那么不允许随机访问。

- 为 `FETCH` 和 `MOVE` 实现了SQL兼容的选项 `FIRST` 、 `LAST` 、 `ABSOLUTE` `_n_` 、 `RELATIVE` `_n_` (Tom)

- 允许在 `DECLARE CURSOR` 上 `EXPLAIN` (Tom)

- 缺省允许 `CLUSTER` 使用标记为先于集群的索引 (Alvaro Herrera)

- 允许 `CLUSTER` 集群所有的表 (Alvaro Herrera)

  这允许一个数据库中所有以前集群的表用一个命令就能重新集群。

- 阻止在局部索引上 `CLUSTER` (Tom)

- 在 `COPY` 文件中允许DOS和Mac行尾结束符 (Bruce)

- 不允许文字的回车作为一个数据值，仍然允许反斜杠回车和 `\r` (Bruce)

- `COPY` 改变 (二进制, `\.`) (Tom)

- 干净的从 `COPY` 失败中恢复 (Tom)

- 阻止 `COPY` 中可能的内存泄露 (Tom)

- 使得 `TRUNCATE` 事务安全 (Rod)

  `TRUNCATE` 现在可以在一个事务内部使用了。如果该事务退出， `TRUNCATE` 所做的改变自动回滚。

- 允许像 `FETCH` 和 `EXPLAIN` 这样的实用命令预备/捆绑 (Tom)
- 添加了 `EXPLAIN EXECUTE` (Neil)
- 通过减少WAL流量提高索引上的 `VACUUM` 性能 (Tom)
- 功能性索引已经泛化为表达式上的索引 (Tom)

  在以前的版本中，功能性索引只支持一个简单的应用于一个或更多字段名的功能。 这个版本允许任意类型的标量表达式。
- 让 `SHOW TRANSACTION ISOLATION` 匹配输入到 `SET TRANSACTION ISOLATION` (Tom)
- 让 `COMMENT ON DATABASE` 在非局部的数据库上生成一个警告，而不是一个错误 (Rod)

  数据库注释存储于数据库本地表中，所以在数据库上的注释必须存储于每个数据库中。
- 提高 `LISTEN` / `NOTIFY` 的可靠性 (Tom)
- 允许 `REINDEX` 可靠的重建非共享的系统目录索引 (Tom)

  这允许系统表不用单独会话的要求就能重建索引，而这在以前的版本中是必须的。 现在需要单独会话重建索引的仅有的表是全局系统表 `pg_database` 、 `pg_shadow` 和 `pg_group` 。

E.183.3.7. 数据类型和函数的变化

- 新增服务器参数 `extra_float_digits` ， 控制浮点数的精度显示 (Pedro Ferreira, Tom)

  控制引起回归测试问题的输出精度。
- 允许 `+1300` 作为一个数字时区说明符，FJST (Tom)
- 删除很少使用的功能 `oidrand` 、 `oidsrand` 和 `userfntest` (Neil)
- 添加 `md5()` 函数到主服务器，已经存在于 `contrib/pgcrypto` 中 (Joe)

  经常请求MD5函数。对于更复杂的加密要求，使用 `contrib/pgcrypto` 。
- 增加 `timestamp` 的数据范围 (John Cochran)
- 更改 `EXTRACT(EPOCH FROM timestamp)` ， 这样 `timestamp without time zone` 假设为本地时间，而不是GMT (Tom)
- 在操作系统不阻止除以零的情况下阻止被零除 (Tom)
- 改变 `numeric` 数据类型内部基于10000 (Tom)
- 新增 `hostmask()` 函数 (Greg Wickham)

- 修复 `to_char()` 和 `to_timestamp()` (Karel)
- 允许函数接受任意参数数据类型并返回任意数据类型，使用 `anyelement` 和 `anyarray` (Joe)

  这允许函数的创建可以与任意数据类型一起工作。

- 数组现在可以指定为 `ARRAY[1,2,3]` 、 `ARRAY[['a','b'],['c','d']]` 或 `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- 允许适当的比较数组，包括 `ORDER BY` 和 `DISTINCT` 支持 (Joe)
- 允许在数组字段上索引 (Joe)
- 允许用 `||` 串联数组 (Joe)
- 允许 `WHERE` 限制 `_expr_` `_op_` ANY/SOME/ALL (`_array_expr_`) (Joe)

  这允许数组表现的像值的列表，对于像 `SELECT * FROM tab WHERE col IN (array_val)` 这样的目的。

- 新增数组函数 `array_append` , `array_cat` , `array_lower` , `array_prepend` , `array_to_string` , `array_upper` , `string_to_array` (Joe)
- 允许用户定义的聚集使用多态函数 (Joe)
- 允许分配给空数组 (Joe)
- 允许60在 `time` 、 `timestamp` 和 `interval` 输入值的秒字段 (Tom)

  闰秒需要60秒。

- 允许 `cidr` 数据类型被转换为 `text` (Tom)
- 不允许无效的时区名出现在SET TIMEZONE中
- 当 `char` 转换为 `varchar` 或 `text` 时，去掉尾随的空白 (Tom)
- 使得 `float(``_p_` )以二进制数字测量精度 `_p_` ， 而不是十进制数字 (Tom)
- 添加IPv6支持到 `inet` 和 `cidr` 数据类型 (Michael Graff)
- 添加 `family()` 函数报告地址是IPv4还是IPv6 (Michael Graff)
- 让 `SHOW datestyle` 产生的输出类似于 `SET datestyle` 使用的 (Tom)
- 让 `EXTRACT(TIMEZONE)` 和 `SET/SHOW TIME ZONE` 遵从SQL时区偏移量符号的惯例，如正数为UTC的东边 (Tom)
- 修复 `date_trunc('quarter', ...)` (Böjthe Zoltán)

以前的版本为这个函数调用返回不正确的值。

- 让 `initcap()` 与Oracle更加兼容 (Mike Nolan)

  `initcap()` 现在大写任何出现在非字母数字字符后面的字母， 不只是在空白后面的字母。

- 日期值只允许 `datestyle` 字段顺序，不以ISO-8601格式 (Greg)
- 添加新的 `datestyle` 值 `MDY` 、 `DMY` 和 `YMD` 来设置输入字段顺序； 遵从 `US` 和 `European` 的向后兼容性 (Tom)
- 像 `'now'` 和 `'today'` 这样的字符串字面值不再作为一个字段的缺省。 使用 `now()` 、 `current_timestamp` 这样的函数代替。（改变需要预备语句） (Tom)
- 在 `min()` / `max()` 中将NaN看做大于任意其他值 (Tom)

  对于大多数目的，NaN早已排序在普通数值的后面，但是 `min()` 和 `max()` 没有获得这个权利。

- 阻止间隔抑制 `:00` 秒的显示
- 新增函数 `pg_get_triggerdef(prettyprint)` 和 `pg_conversion_is_visible()` (Christopher)
- 允许指定时间为 `040506` 或 `0405` (Tom)
- 输入日期顺序现在必须为 `YYYY-MM-DD` （年为4位数）或匹配 `datestyle`
- 让 `pg_get_constraintdef` 支持唯一、主键和检查约束 (Christopher)

E.183.3.8. 服务器端语言的变化

- 当 `RETURN NEXT` 用在零行字段变量上时阻止PL/pgSQL崩溃 (Tom)
- 让PL/Python的 `spi_execute` 接口正确的处理空值 (Andrew Bosma)
- 允许PL/pgSQL不用 `%ROWTYPE` 声明复合类型的变量 (Tom)
- 修复PL/Python的 `_quote()` 函数，以处理大的整数
- 使PL/Python语言不可信，现在称为 `plpythonu` (Kevin Jacobs, Tom)

  Python语言不再支持受限制的执行环境，所以删除了受信任的PL/Python版本。 如果这个情况改变了，将阅读非超级用户可以使用的PL/Python的一个版本。

- 允许多态的PL/Python函数 (Joe, Tom)
- 允许多态的SQL函数 (Joe)
- 用对多态性的全面支持改进PL/pgSQL中编译的函数缓存机制 (Joe)

- 在PL/pgSQL中添加新的参数 `$0`，表示该函数的实际返回类型 (Joe)
- 允许PL/Tcl和PL/Python在多重表上使用相同的触发器 (Tom)
- 修复PL/Tcl的 `spi_prepare` 以在参数类型列表中接受完全限定的类型名 (Jan)

E.183.3.9. psql的变化

- 添加 `\pset pager always` 以总是使用pager (Greg)

  强制使用pager，即使行数少于屏幕高度也是一样。这个对于有几个屏幕的行有帮助。
- 改善tab补齐 (Rod, Ross Reedstrom, Ian Barwick)
- 重新排序 `\?` 帮助分组 (Harald Armin Massa, Bruce)
- 添加反斜杠命令列出模式、计算和转换 (Christopher)
- `\encoding` 现在根据服务器参数 `client_encoding` 变化 (Tom)

  在以前的版本中， `\encoding` 不知道使用 `SET client_encoding` 所做的编码改变。
- 将编辑器缓冲区保存到readline历史 (Ross)

  当使用 `\e` 编辑一个查询时，结果被保存到readline历史，以使用向上箭头键检索。
- 改善 `\d` 的显示 (Christopher)
- 提高HTML模式，使其更加符合标准 (Greg)
- 新增 `\set AUTOCOMMIT off` 能力 (Tom)

  这取代了删除的服务器参数 `autocommit` 。
- 新增 `\set VERBOSITY` 控制错误详情 (Tom)

  这个控制新的错误报告细节。
- 新增提示转义序列 `%x` ，显示事务状态 (Tom)
- psql的长选项现在在所有平台上可用

E.183.3.10. pg_dump的变化

- 多重pg_dump修复，包括tar格式和大对象
- 允许pg_dump转储指定的模式 (Neil)
- 让pg_dump保存字段存储特性 (Christopher)

  这保存了 `ALTER TABLE ... SET STORAGE` 信息。

- 使pg_dump保存 `CLUSTER` 特性 (Christopher)
- 让pg_dumpall使用 `GRANT` / `REVOKE` 转储数据库级别的权限 (Tom)
- 允许pg_dumpall支持pg_dump的选项 `-a` 、 `-s` 、 `-x` (Tom)
- 阻止pg_dump小写在命令行中指定的标识符 (Tom)
- pg_dump选项 `--use-set-session-authorization` 和 `--no-reconnect` 现在什么也不做， 所有转储使用 `SET SESSION AUTHORIZATION`

  pg_dump不再重新连接到切换的用户，但是反而总是使用 `SET SESSION AUTHORIZATION` 。 这将减少在恢复时的口令提示。

- pg_dump的长选项现在在所有平台上可用

  PostgreSQL现在包括它自己的长选项处理例程。

E.183.3.11. libpq的变化

- 添加函数 `PQfreemem` 在Windows上释放内存， 建议 `NOTIFY` (Bruce)

  Windows要求在库中分配的内存通过在相同库中的一个函数释放，因此 `free()` 并不释放libpq分配的内存。 `PQfreemem` 是释放libpq内存的适当的方式， 尤其是在Windows上，也推荐在其他平台上使用。

- 文档服务能力，并且添加了样本文件 (Bruce)

  这允许客户查看客户端机器上中心文件的连接信息。

- 让 `PQsetdbLogin` 有和 `PQconnectdb` 相同的缺省 (Tom)
- 当结果集太大时，允许libpq彻底的失败 (Tom)
- 提高函数 `PQunescapeBytea` 的性能 (Ben Lamb)
- 允许线程安全的libpq带有 `configure` 选项 `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- 允许函数 `pqInternalNotice` 接受一个格式字符串和参数， 而不只是一个预定义格式的信息 (Tom, Sean Chittenden)
- 用 `sslmode` 值 `disable` 、 `allow` 、 `prefer` 和 `require` 控制SSL协商 (Jon Jensen)
- 允许新的错误代码和文本级别 (Tom)
- 允许访问基础表和查询结果的字段 (Tom)

  这对于想要知道和指定结果集相关的基础表和字段的查询生成器应用是有用的。

- 允许访问当前事务状态 (Tom)
- 添加传递二进制数据直接到服务器的能力 (Tom)
- 添加函数 `PQexecPrepared` 和 `PQsendQueryPrepared`， 执行以前预备语句的绑定/执行 (Tom)

E.183.3.12. JDBC的变化

- 允许在可更新的结果集上 `setNull`
- 允许在预备语句上 `executeBatch` (Barry)
- 支持SSL连接 (Barry)
- 在结果集中处理模式名 (Paul Sorenson)
- 添加refcursor支持 (Nic Ferrier)

E.183.3.13. 其他接口的变化

- 在libpgtcl关闭期间阻止可能的内存泄露或内核转储 (Tom)
- 添加到ECPG的Informix兼容性 (Michael)

  这允许ECPG处理用某些Informix扩展写的嵌入的C程序。
- 为Informix添加定长的类型 `decimal` 到ECPG (Michael)
- 允许线程安全的嵌入式SQL程序带有 `configure` 选项 `--enable-thread-safety` (Lee Kindness, Bruce)

  这允许同一时间多线程访问数据库。
- 移动Python客户端PyGreSQL到http://www.pygresql.org (Marc)

E.183.3.14. 源代码的变化

- 阻止对单独平台几何回归结果文件的需要 (Tom)
- 改进PPC的锁定原语 (Reinhard Max)
- 新增函数 `palloc0` 分配和清理内存 (Bruce)
- 为s390x CPU (64位)修复锁定代码 (Tom)
- 允许OpenBSD使用用局部鉴别凭证 (William Ahern)
- 使查询规划树对于执行者是只读的 (Tom)

- 添加Darwin启动脚本 (David Wheeler)
- 允许libpq用Borland C++编译器编译 (Lester Godwin, Karl Waclawek)
- 如果需要，使用我们自己的 `getopt_long()` 版本 (Peter)
- 转换管理脚本为C (Peter)
- 如果从CVS编译，Bison >= 1.85现在需要建立PostgreSQL语法
- 合并文档到一本书中 (Peter)
- 添加Windows兼容性函数 (Bruce)
- 允许客户端接口在MinGW下编译 (Bruce)
- 为错误报告新增 `ereport()` 函数 (Tom)
- 在Linux上支持Intel编译器 (Peter)
- 改善Linux启动脚本 (Slawomir Sudnik, Darko Prenosil)
- 添加对AMD Opteron和Itanium的支持 (Jeffrey W. Baker, Bruce)
- 从 `configure` 中删除 `--enable-recode` 选项

  不再需要这个了，我们有了 `CREATE CONVERSION` 。
- 如果没有找到自旋锁代码，那么生成一个编译错误 (Bruce)

  没有自旋锁代码的平台现在将不能编译，而不是默默的使用信号灯。 这个失败可以使用一个新的 `configure` 选项禁用。

E.183.3.15. 贡献包的变化

- 改变dbmirror许可为BSD
- 改善earthdistance (Bruno Wolff III)
- 提高pgcrypto的可移植性 (Marko Kreen)
- 阻止xml中的崩溃 (John Gray, Michael Richards)
- 更新oracle
- 更新mysql
- 更新cube (Bruno Wolff III)
- 更新earthdistance以使用cube (Bruno Wolff III)

- 更新btree_gist (Oleg)
- 新增tsearch2全文本搜索模块 (Oleg, Teodor)
- 添加基于散列的交叉表函数到tablefuncs (Joe)
- 添加连续的字段以排序tablefuncs中的`connectby()`兄弟姐妹 (Nabil Sayegh,Joe)
- 添加命名的持久连接到dblink (Shridhar Daithanka)
- 新的pg_autovacuum允许自动`VACUUM` (Matthew T. O'Connor)
- 让pgbench遵守环境变量`PGHOST`、`PGPORT`、`PGUSER` (Tatsuo)
- 改进intarray (Teodor Sigaev)
- 改进pgstattuple (Rod)
- 修复fuzzystrmatch中`metaphone()`的错误
- 改进adddepend (Rod)
- 更新spi/timetravel (Böjthe Zoltán)
- 修复dbase `-s` 选项并改进非ASCII的处理 (Thomas Behr, Márcio Smiderle)
- 删除数组模块，因为现在默认情况下包括数组特性 (Joe)

E.184. 版本 7.3.21

> 发布日期: 2008-01-07

这个版本包含7.3.20的各种补丁，包括重大安全问题的补丁。

预计这是7.3.X系列的最后一个PostgreSQL版本。鼓励用户尽快升级到一个新的版本分支。

E.184.1. 迁移到版本 7.3.21

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.184.2. 修改列表

- 阻止用户的权限运行 `VACUUM`，`ANALYZE` 等执行索引里的函数（Tom）。

  应用于索引表达式和部分索引谓词的函数在建立一个新的表项时被评估。一直以来以为， 如果一个人修改了一个未被信任的用户的表，这造成了特洛伊木马代码扩展的风险。（请注意，触发器、缺省、检查约束等造成相同类型的风险。） 但是索引里的函数造成额外的威胁，因为他们将被日常维护操作如 `VACUUM FULL` 执行，这些操作通常由一个超级用户账户自动执行。例如， 一个不法用户可以通过设置一个特洛伊木马索引定义并等待下一个日常清理，用超级用户权限执行代码。 修复标准维护操作（包括 `VACUUM`，`ANALYZE`，`REINDEX`，和 `CLUSTER`）的安排为作为表的所有者执行而不是调用用户， 使用 `SECURITY DEFINER` 函数已经使用的相同的权限切换机制。 为了阻止绕过这个安全措施，现在禁止 `SET SESSION AUTHORIZATION` 和 `SET ROLE` 在 `SECURITY DEFINER` 环境中执行了。(CVE-2007-6600)

- 需要使用 `/contrib/dblink` 的非超级用户只使用口令认证，作为一个安全措施(Joe)。

  在7.3.20里出现的修复是不完整的，因为它为了一些 `dblink` 函数堵塞了整个。 (CVE-2007-6601, CVE-2007-3278)

- 当使用一个多字节数据库编码时，修复 `translate()` 里的潜在冲突。(Tom)

- 使得 `contrib/tablefunc` 的 `crosstab()` 处理空rowid本身为一个类别， 而不是冲突。(Joe)

- 当重新生成 `configure` 脚本时需要使用一个特定的Autoconf版本。(Peter)

  这只影响开发者和打包者。这样做是为了阻止意外的使用未经检验的Autoconf 和PostgreSQL版本的组合。如果你真的想要使用一个不同的Autoconf 版本，你可以移除版本检查，但是结果如何你自己负责。

E.185. 版本 7.3.20

> 发布日期: 2007-09-17

这个版本包含7.3.19的补丁。

E.185.1. 迁移到版本 7.3.20

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来，请查阅7.3.13的版本说明。

E.185.2. 修改列表

- 当一个事务插入行然后在接近相同表上当前 `VACUUM` 的结尾时终止时，阻止索引损坏。（Tom）
- 使 `CREATE DOMAIN ... DEFAULT NULL` 正确工作（Tom）
- 修复当 `log_min_error_statement` 日志耗尽内存时的崩溃（Tom）
- 需要使用 `/contrib/dblink` 的非超级用户使用唯一口令认证，作为安全措施(Joe)

E.186. 版本 7.3.19

> 发布日期: 2007-04-23

这个版本包含7.3.18的补丁，包括一个安全补丁。

E.186.1. 迁移到版本 7.3.19

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.186.2. 修改列表

- 支持在`search_path`中对临时表模式明确布局，并且禁用搜索函数和操作符(Tom)

  这需要允许安全定义函数设置`search_path`的安全值。没有这个， 一个非特权SQL用户可以使用临时对象用安全定义函数的权限执行代码(CVE-2007-2138)。 参阅`CREATE FUNCTION`获取更多信息。

- 修复`VACUUM FULL`处理`UPDATE`链时的潜在数据损坏臭虫(Tom, Pavan Deolasee)

E.187. 版本 7.3.18

> 发布日期: 2007-02-05

这个版本包含各种7.3.17的补丁，包括安全补丁。

E.187.1. 迁移到版本 7.3.18

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.187.2. 修改列表

- 移除了允许连接用户读取后端存储的安全漏洞(Tom)

  安全隐患包括在一个SQL函数中改变表字段的数据类型(CVE-2007-0555)。 这个错误很容易被利用导致后端崩溃，并且原理上可能被用来读取该用户不能访问的数据库内容。

- 修复在不可行的分解点拆分btree索引页可能失败的罕见bug(Heikki Linnakangas)

- 加强超过三个字节长度的UTF8序列的多类型字符处理的安全(Tom)

E.188. 版本 7.3.17

> 发布日期: 2007-01-08

这个版本包含各种7.3.16的补丁。

E.188.1. 迁移到版本 7.3.17

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.188.2. 修改列表

- 对于新initdb安装来说， `to_number()` 和 `to_char(numeric)` 现在是 `STABLE` 而不是 `IMMUTABLE` (Tom)

  这是因为 `lc_numeric` 可以潜在的改变这些函数的输出。

- 改善使用圆括号的正则表达式的索引使用(Tom)

  也改善psql `\d` 的性能

E.189. 版本 7.3.16

> 发布日期: 2006-10-16

这个版本包含各种7.3.15的补丁。

E.189.1. 迁移到版本 7.3.16

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.189.2. 修改列表

- 修复模式中匹配psql的 `\d` 命令的极端情况。
- 修复/contrib/ltree里的索引损坏bug(Teodor)
- 移植 7.4 自旋锁代码以提升性能和更好的支持64位架构
- 修复libpq中的SSL相关的内存泄露
- 修复 /contrib/dbmirror中的反斜杠逃逸
- 为最近US DST 法律的变化调整回归测试

E.190. 版本 7.3.15

> 发布日期: 2006-05-23

这个版本包含各种7.3.14的补丁，包括极端严重的安全问题的补丁。

E.190.1. 迁移到版本 7.3.15

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

在CVE-2006-2313 和 CVE-2006-2314中描述的针对SQL注入攻击的完全安全可能需要在应用代码中改变。 如果你的应用在SQL命令中嵌入了不能信任的字符串，那么你应该尽快检查它们， 以确保它们正在使用推荐的转义技术。在大多数情况下，应用应该使用库或驱动提供的子程序 （如libpq的 `PQescapeStringConn()` ）来执行字符串转义， 而不是依赖于*ad hoc*代码。

E.190.2. 修改列表

- 更改服务器使在其任何情况下都拒绝无用编码的多字节字符(Tatsuo, Tom)

  当PostgreSQL在某段时间一直朝着这个方向移动时， 检查现在一致的应用到所有的编码和所有的文本输入，并且现在错误不仅仅是警告那么简单。 这个改变防范在CVE-2006-2313里描述的SQL注入攻击类型。

- 拒绝字符串里不安全的使用 `\'`

  作为防范在CVE-2006-2314里描述的SQL注入攻击类型的服务器端，服务器现在只接受 `''` 而不是 `\'` 作为SQL字符串字面值里的ASCII单引号表示。缺省的， 只有 `client_encoding` 设置为仅客户端的编码(SJIS, BIG5, GBK, GB18030, 或 UHC)时拒绝 `\'` ，这是可能有SQL注入的情况。 一个新的配置参数 `backslash_quote` 可在需要时用于调整这个行为。 请注意，针对CVE-2006-2314的完全安全可能需要客户端侧的改变； `backslash_quote` 的目的是在某种程度上使不安全的客户端是不安全的显而易见。

- 修改libpq的字符串逃逸例程，使其知道编码注意事项

  libpq使用这个补丁应用于CVE-2006-2313 和 CVE-2006-2314中描述的安全问题。 使用多个PostgreSQL并发连接的应用应该迁移到 `PQescapeStringConn()` 和 `PQescapeByteaConn()` ，以确保为每个数据库连接中使用的设置正确的转义。 "手动"进行字符串转义的应用应该被修改为依赖于库例程。

- 修复一些不正确的编码转换函数

`win1251_to_iso`，`alt_to_iso`，`euc_tw_to_big5`，`euc_tw_to_mic`，`mic_to_euc_tw` 都分配为不同的范围。

- 清理字符串中残留的 `\'` 的使用(Bruce, Jan)
- 正确的修复服务器使用自定义的DH SSL参数(MichaelFuhr)
- 修复各种小内存泄露

E.191. 版本 7.3.14

> 发布日期: 2006-02-14

这个版本包含各种7.3.13的补丁。

E.191.1. 迁移到版本 7.3.14

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.13更早的版本升级而来， 请查阅7.3.13的版本说明。

E.191.2. 修改列表

- 修复 `SET SESSION AUTHORIZATION` (CVE-2006-0553)中潜在的危机

  非特权的用户可能使服务器进程崩溃，导致短暂的拒绝服务其他用户，如果服务器启用维护编译 （非缺省的）。感谢Akio Ishida报告了这个问题。

- 修复自插入行的行可见逻辑的bug(Tom)

  在罕见情况下，通过当前命令插入的行会被看做早已经有效，此时它不应该有效。 修复bug在7.3.11版本中创建。

- 修复在pg_clog文件创建时会导致"文件已经存在"错误的竞态条件。

- 修复以允许有交叉模式的恢复转储引用自定义操作符(Tom)

- 在配置期间测试 `finite` 和 `isinf` 的存在的可移植性修复(Tom)

E.192. 版本 7.3.13

> 发布日期: 2006-01-09

这个版本包含各种7.3.12以来的补丁。

E.192.1. 迁移到版本 7.3.13

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.10更早的版本升级而来， 请查阅7.3.10的版本说明。 另外，如果你被下面描述的本地或plperl问题影响， 你可能需要在升级之后 `REINDEX` 在文本字段上的索引。

E.192.2. 修改列表

- 为本地认为不同字符组合平等的想法修复字符串比较，如Hungarian (Tom)

  这可能需要 `REINDEX` 来修复在文本字段上的现有索引。

- 在主机启动期间设置本地环境变量，以确保plperl稍后不会更改本地设置

  这修复了postmaster启动时不使用initdb指定的环境变量的问题。 在这种情况下，plperl的任何使用都可能导致损坏索引。 如果发生了这个，可能需要 `REINDEX` 来修复在文本字段上的现有索引。

- 修复在strpos()和在某些很少使用的Asian多字节字符设置中处理的正则表达式中长期存在的bug(Tatsuo)

- 修复在 `/contrib/pgcrypto` gen_salt中的bug， 该bug导致它不为MD5或XDES算法使用所有可用的salt空间(Marko Kreen, Solar Designer)

  salt对Blowfish和标准DES没有影响。

- 当指定的字段号不同于查询实际返回的值时，修复 `/contrib/dblink` 抛出一个错误，而不是死机(Joe)

E.193. 版本 7.3.12

> 发布日期: 2005-12-12

这个版本包含各种自7.3.11以来的补丁。

E.193.1. 迁移到版本 7.3.12

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.10更早的版本升级而来， 请查阅7.3.10的版本说明。

E.193.2. 修改列表

- 修复事务日志管理中的竞态条件

  这里有一个窗口，I/O操作可以从错误页开始，导致维护失败或数据损坏。

- 修复了 `/contrib/ltree` (Teodor)
- 修复了对于外连接长期存在的规划错误

  这个bug有时会导致一个假的错误"右连接只支持合并可连接的连接条件"。

- 当一个表已经被删除时，阻止pg_autovacuum里的核心转储。

E.194. 版本 7.3.11

> 发布日期: 2005-10-04

这个版本包含自7.3.10以来的各种补丁。

E.194.1. 迁移到版本 7.3.11

运行7.3.X的不需要转储/恢复。不过，如果从一个7.3.10更早的版本升级而来， 请查阅7.3.10的版本说明。

E.194.2. 修改列表

- 修复允许 `VACUUM` 移除 `ctid` 链太快的错误， 并在代码中添加更多跟随 `ctid` 连接的检查

  这修复了在十分罕见的情况下可能导致死机的长期存在的问题。

- 当使用多字节字符设置时，修复 `CHAR()` 使其适当添加空白到指定的长度(Yoshiyuki Asaba)

  在先前的版本中， `CHAR()` 的填充是不正确的， 因为它只填充指定的字节数而不考虑存储了多少个字符。

- 修复在类似 `UPDATE a=... WHERE a...` with GiST index on column `a` 的查询中丢失行的问题
- 改善部分写入WAL页面的检查
- 改善启用SSL时信号处理的健壮性
- 修复了各种内存泄露
- 各种可移植性改善
- 修复当变量类型是通过参数传递时，PL/pgSQL正确处理 `var := var` 的问题。

E.195. 版本 7.3.10

> 发布日期: 2005-05-09

这个版本包含各种自7.3.9以来的补丁，包括几个安全相关的问题。

E.195.1. 迁移到版本 7.3.10

运行7.3.X的不需要转储/恢复。不过， 转储/恢复是处理在7.3.X系统目录中的初始化内容中发现的重大安全问题的一种方式。 使用7.3.10的initdb的dump/initdb/reload序列将自动改正这个问题。

安全问题是非特权用户可以通过SQL命令引用内建字符设置编码转换函数， 但是该函数不是这样使用的，并且对于恶意的参数选择是不安全的。 修复包括改变这些函数声明的参数列表，这样它们可以不再被SQL命令调用。 （这样并不影响它们通过编码转换机制的正常使用。） 强烈推荐所有安装修复这个错误，通过initdb或通过下面给出的手动修复过程。 该错误至少允许非特权数据库用户毁坏他们的服务器进程， 甚至可能允许非特权用户获得数据库超级用户的权限。

如果你希望不做initdb，那么执行下列的过程。作为数据库超级用户，执行：

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
      AND proargtypes[2] = 'cstring'::regtype;
-- 该命令应该报告已经更新了90行;
-- 如果不是，那么回滚并调查而不是提交！
COMMIT;
```

上面的程序必须在每个安装的数据库中执行，包括 `template1` ， 并且理论上也包括 `template0` 。如果你不修复模板数据库， 那么任何随后创建的数据库将包含相同的错误。 `template1` 可以用与任意其他数据库相同的方式修复，但是修复 `template0` 需要额外的步骤。首先，从任意数据库发出：

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

然后连接到 `template0` 并执行以上所述的修复程序。最后，执行：

```
-- 重新冻结template0:
VACUUM FREEZE;
-- 保护它免受未来的变化:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.195.2. 修改列表

- 改变编码函数签名以阻止滥用
- 修复古老竞态条件，该条件允许一个事务因为某些目的(如 SELECT FOR UPDATE)被视作已提交

  这是一个极其严重的bug，因为它会导致表面数据的不一致性被应用短暂的看到。
- 修复相关扩展和VACUUM之间的竞态条件

  这理论上可能导致丢失新近插入的数据的一个页面，尽管情况看起来概率很低。 没有导致多于一个维护失败的已知情况。
- 修复了 `TIME WITH TIME ZONE` 值的比较

  当使用 `--enable-integer-datetimes` 配置开关时，该比较代码是错误的。 注意：如果你在 `TIME WITH TIME ZONE` 字段上有索引， 它将需要在安装这个更新后 `REINDEX`，因为这个修复纠正了字段值的顺序。
- 为 `TIME WITH TIME ZONE` 值修复了 `EXTRACT(EPOCH)`
- 修复了 `INTERVAL` 值中负分数秒的错误显示

  这个错误只有在使用了 `--enable-integer-datetimes` 配置开关的时候发生。
- 在plpgsql中额外的缓冲区溢出检查(Neil)
- 修复pg_dump转储触发器名字正确的包含 `%` (Neil)
- 阻止 `to_char(interval)` 为月份相关的格式转储内核
- 为更新的OpenSSL构建修复 `contrib/pgcrypto` (Marko Kreen)
- 为 `contrib/intagg` 修复更多的64位问题
- 阻止返回 `RECORD` 的函数的不正确的最优化

E.196. 版本 7.3.9

> 发布日期: 2005-01-31

这个版本包含自7.3.8以来的各种补丁，包括几个安全相关的问题。

E.196.1. 迁移到版本 7.3.9

运行7.3.X的不需要转储/恢复。

E.196.2. 修改列表

- 不允许 `LOAD` 到非超级用户

  在平台上将自动执行共享库（至少包括窗口和ELF-based Unixen）的初始化函数，`LOAD` 可以用来使服务器执行任意的代码。感谢NGS Software报告这个问题。

- 检查聚集函数的创建者有执行指定转换函数的权利

  这个疏忽使其可以绕过函数上EXECUTE权限的拒绝。

- 修复 contrib/intagg 中的安全和64位问题
- 给某些contrib函数添加需要的STRICT标记(Kris Jurka)
- 当plpgsql游标声明有太多的参数时避免缓冲区溢出(Neil)
- 为FULL 和 RIGHT外连接修复规划错误

  连接的结果被错误的认为和左输入的排序相同。这不止会发送错误排序的输出给用户，而且在嵌套合并连接的情况下会给出完全错误的回复。

- 修复plperl在元组字段的双引号标记
- 修复SQL和GERMAN数据类型中负数间隔的显示

E.197. 版本 7.3.8

> 发布日期: 2004-10-22

这个版本包含各种自7.3.7以来的补丁。

E.197.1. 迁移到版本 7.3.8

运行7.3.X的不需要转储/恢复。

E.197.2. 修改列表

- 修复可能未能更新到磁盘上的提示

  在很少的情况下，这个疏忽会导致"不能访问事务状态"错误， 这是一个潜在的数据丢失bug。

- 确保散列的外连接不丢失元组

  使用哈希连接规划的非常大的左连接不会输出不匹配的左侧行只给出正确的数据分布。

- 不允许作为root运行pg_ctl

  这是为了预防任何可能的安全问题。

- 避免在make_oidjoins_check中使用 /tmp 中的临时文件

  这已经作为一个安全问题报告了，尽管它几乎不值得关心，因为没有原因会使非开发者使用这个脚本。

E.198. 版本 7.3.7

> 发布日期: 2004-08-16

这个版本包括7.3.6的一个关键的修复，和一些小的项目。

E.198.1. 迁移到版本7.3.7

运行7.3.X的不需要转储/恢复。

E.198.2. 修改列表

- 在死机期间阻止可能的已提交事务的丢失

  由于事务提交和检查点之间不充分的连锁，正好在最近的检查点之前提交的事务可能会丢失， 全部的或部分的，在数据库崩溃和重启后。这是一个自PostgreSQL 7.1 以来就存在的严重bug。

- 删除tsearch中不对称的文字处理(Teodor)

- 在pg_dump一个CAST时，恰当的模式限定函数名

E.199. 版本 7.3.6

> 发布日期: 2004-03-02

这个版本包含各种自7.3.5以来的补丁。

E.199.1. 迁移到版本 7.3.6

运行 7.3.* 版本的用户不需要转储/恢复。

E.199.2. 修改列表

- 修复规则权限检查中错误的改变

  在7.3.3中应用的补丁修复一个极端情况，在许多不是很极端的情况下， 规则权限检查变成有禁用规则相关权限的检查。这可能比如允许用户插入到他们不希望有权限插入的视图中。 我们因此恢复7.3.3补丁。原始的bug将在8.0中修复。

- 修复GetNewTransactionId()中操作的不正确的顺序

  这个bug会导致超出磁盘空间条件的失败，包括没有重启的能力，即使磁盘空间空闲之后也不行。

- 即使提供了一个额外的值给CFLAGS也要确保配置选择 fno 严格的别名。

  在某些平台上，用-fstrict-aliasing建立会导致bug。

- 使 pg_restore 正确的处理64位的 off_t

  这个bug阻止从超过4 GB的归档文件中正确的恢复。

- 使 contrib/dblink 不假设本地和远程类型OID匹配(Joe)

- 恰当的引用connectby()的start_with参数(Joe)

- 当plpgsql函数的行类型参数是NULL时不死机

- 当规划LIKE操作符时避免在极端情况下产生无效字符编码序列

- 确保text_position()在多字节情况下不能扫描通过源字符串的结尾(Korea PostgreSQL Users' Group)

- 修复索引优化并选择性的为LIKE操作符的bytea字段估计(Joe)

E.200. 版本 7.3.5

> 发布日期: 2003-12-03

有各种自7.3.4以来的补丁。

E.200.1. 迁移到版本 7.3.5

运行 7.3.* 版本的用户不需要转储/恢复。

E.200.2. 修改列表

- 强制零损坏的页面在WAL恢复期间工作
- 阻止某些"变量不在子计划目录列表"的模糊情况
- 强制统计过程从共享内存中分离，确保清理关闭
- 使PQescapeBytea 和 byteaout相互一致(Joe)
- 添加了丢失的SPI_finish()调用到dblink的get_tuple_of_interest() (Joe)
- 当规则重写INSERT时修复可能的违反外键约束(Jan)
- 在PL/Tcl的spi_prepare命令中支持合格的类型名(Jan)
- 使pg_dump处理pg_catalog中的过程语言处理器
- 使pg_dump处理自定义操作符类是另外一个模式的情况
- 使pg_dump正确的转储二进制兼容的转换
- 修复包含子查询的表达式插入到规则中
- 修复clusterdb脚本中不正确的参数处理(Anand Ranganathan)
- 修复plpython触发器中已删除字段的问题
- 修复to_char()读取过去的输入字符串结束的问题(Karel)
- 修复GB18030映射错误(Tatsuo)
- 修复几个SSL错误处理和异步SSL I/O问题
- 修复在JDBC中绑定一个值的列表到单个参数的能力(阻止可能的 SQL 注入攻击)

- 修复HAVE_INT64_TIMESTAMP代码路径中的一些错误
- 修复btree索引和第一次根页面分离并行的极端情况

E.201. 版本 7.3.4

> 发布日期: 2003-07-24

有各种自7.3.3以来的补丁。

E.201.1. 迁移到版本 7.3.4

运行 7.3.* 版本的用户不需要转储/恢复。

E.201.2. 修改列表

- 修复2000以前的timestamp-to-date日期转换的损坏
- 阻止极少服务器启动失败的可能性(Tom)
- 修复interval-to-time转换中的bug(Tom)
- 在pg_dump中的几个地方添加约束名(Rod)
- 提高有很多参数的函数的性能(Tom)
- 修复to_ascii()缓冲溢出(Tom)
- 阻止抛出一个恢复数据库评论的错误(Tom)
- 解决存在于一些Solaris版本中的多个问题的strxfrm()(Tom)
- 恰当的逃逸jdbc setObject()字符串以提高安全性(Barry)

E.202. 版本 7.3.3

> 发布日期: 2003-05-22

这个版本包含版本7.3.2的各种补丁。

E.202.1. 迁移到版本 7.3.3

运行 7.3.* 版本的用户不需要转储/恢复。

E.202.2. 修改列表

- 修复崩溃后计算StartUpID有时不正确的问题
- 避免在一个事务中大量推迟的触发器迟钝(Stephan)
- 当 `UPDATE` 不改变外键的值时不要锁住引用的行(Jan)
- 在Sparc上使用 `-fPIC` 而不是 `-fpic` (Tom Callaway)
- 修复contrib/reindexdb中缺少模式意识的问题
- 修复零元素结果数组的contrib/intarray错误(Teodor)
- 确保createuser脚本在control-C时退出(Oliver)
- 修复删除的字段类型本身已经被删除的错误
- `CHECKPOINT` 在非关键的步骤下不会在错误时引起数据库恐慌
- 接受时间戳、时间、 间隔输入值的秒字段为60
- 如果 `TIMESTAMP` , `TIME` , 或 `INTERVAL` 精度太大， 发出一个通知而不是错误。
- 修复 `abstime-to-time` 转换函数 （initdb后生效）
- 为 `timestampt_izone` 修复pg_proc条目 （initdb后生效）
- 使 `EXTRACT(EPOCH FROM timestamp without time zone)` 将输入当做本地时间
- 如果时区改变在事务之前那么 `'now'::timestamptz` 给出错误回复
- `HAVE_INT64_TIMESTAMP` 代码为time with timezone写满输入
- 接受 `GLOBAL TEMP/TEMPORARY` 作为 `TEMPORARY` 的一个同义词

- 在外键触发器中避免不正确的模式权限检查失败
- 修复 `SET DEFAULT` 动作中的外键触发器的bug
- 为 `UPDATE` 和 `DELETE` 触发器修复行抓取中不正确的time-qual检查
- 外键子句在 `ALTER TABLE ADD COLUMN` 中被解析但是被忽略
- 修复处理器函数已经存在时createlang脚本损坏的情况
- 修复在pg_dump, COPY, ANALYZE, 其他地方的零字段表的错误行为
- 修复 `func_error()` 在类型名包含'%'时的错误行为
- 修复 `replace()` 在字符串包含'%'时的错误行为
- 规则表达式模式包含确定的多字节字符失败
- 在大多数情况下的连接大小估计上为 `NULL` 正确的解释
- 避免 `isblank()` 函数或宏的系统定义冲突
- 修复EUC_TW转换中转换大代码点值的失败
- 修复 `SSL_read` / `SSL_write` 调用的错误恢复
- 不做强制类型转换表达式的早期常数合并
- 验证在任何页面的页面标题字段紧接阅读
- 修复未命名连接中的未分组变量的不正确检查
- 修复 `to_ascii` 中的缓冲溢出(Guido Notari)
- contrib/ltree 修复 (Teodor)
- 修复机器上字符是无符号的死锁检测中的内核转储
- 避免多种方式索引扫描中耗尽内存（7.3中的bug）
- 修复规划者的选择性估计函数正确的处理域
- 修复dbmirror内存分配bug(Steven Singer)
- 阻止 `ln(numeric)` 中因为舍入错误无限循环
- 如果有多个相等的GROUP BY条目，那么 `GROUP BY` 就会感到困惑
- 修复当继承的 `UPDATE` / `DELETE` 参考另外一个继承的表时的糟糕规划
- 阻止在不完整(部分或非NULL存储)的索引上集群
- 如果服务关闭请求到达时仍然在启动那么在适当的时间处理

- 修复临时索引中的左连接（可以通过向后扫描错过入口）
- 修复postgresql.conf中不正确的处理client_encoding设置(Tatsuo)
- 修复在Async_NotifyHandler运行之后未能响应 `pg_ctl stop -m fast`
- 修复规则包含相同类型的多个声明时的SPI
- 修复规则查询中访问权限的错误类型检查的问题
- 修复 `CREATE RULE` 中 `EXCEPT` 的问题
- 预防删除带有序列字段的临时表的问题
- 修复复杂视图中replace_vars_with_subplan_refs的失败
- 修复regexp在单字节编码里的缓慢(Tatsuo)
- 在 `CREATE CAST` 和 `DROP CAST`
- 接受 `SETOF type[]` ，以前写作 `SETOF _type`
- 修复在过程语言的某些情况下的pg_dump内核转储
- 为了可移植性，强制在pg_dump的输出中使用ISO数据类型(Oliver)
- pg_dump未能处理 `lo_read` 返回的错误(Oleg Drokin)
- pg_dumpall分组失败，分组里面没有成员(Nick Eskelinen)
- pg_dumpall未能识别 --globals-only 开关
- 如果声明了-X disable-triggers，那么pg_restore未能存储二进制大对象
- 修复plpgsql中内部函数内存溢出
- 如果给出错误的参数，那么pltcl的 `elog` 命令转储内核(Ian Harding)
- plpython使用了 `atttypmod` 的错误值(Brad McLean)
- 修复Python接口中布尔值的不正确的引用(D'Arcy)
- 为JDBC添加 `addDataType()` 方法到PGConnection接口
- 为JDBC修复可更新的结果集的各种问题(Shawn Green)
- 为JDBC修复DatabaseMetaData的各种问题(Kris Jurka, Peter Royal)
- 修复JDBC中分析表ACLs的问题
- JDBC中为字符集转换问题提供更好的错误信息

E.203. 版本 7.3.2

> 发布日期: 2003-02-04

这个版本包含自版本7.3.1以来的各种补丁。

E.203.1. 迁移到版本 7.3.2

运行 7.3.* 版本的用户不需要转储/恢复。

E.203.2. 修改列表

- 恢复CREATE TABLE AS / SELECT INTO中OID字段的创建
- 当转储的视图有注释时修复pg_dump内核转储
- 适当的转储DEFERRABLE/INITIALLY DEFERRED约束
- 当子表字段编号与父表不同时辅修UPDATE
- 增加max_fsm_relations的缺省值
- 为单行查询修复游标中向后抓取的问题
- 使SELECT DISTINCT查询上的游标向后抓取时正确的工作
- 修复加载包含contrib/lo用法的pg_dump文件的问题
- 修复所有数字用户名的问题
- 修复libpgtcl断开连接期间可能的内存溢出和内核转储
- 使plpython的spi_execute命令正确的处理null (Andrew Bosma)
- 调整plpython错误报告，使其回归测试再次通过
- 与bison 1.875共事
- 在plpgsql的 %type 中正确的处理大小写混合的名字(Neil)
- 当执行一个由规则写出的查询时修复pltcl中的内核转储
- 修复数组下标溢出(来自 Yichen Xie 的报告)
- 在浮点数情况下将MAX_TIME_PRECISION从13降低到10

- 在per-database 和 per-user设置中正确的包含变量名
- 当SELECT into record不返回行时修复plpgsql的 RETURN NEXT 中的内核转储
- 修复python客户端接口中pg_type.typprtlen的过时的使用
- 正确的处理JDBC驱动中的timestamps中的小数秒
- 提升JDBC中getImportedKeys()的性能
- 使共享库符号链接在HPUX上标准的工作(Giles)
- 为timestamp, time, interval修复不一致的舍入行为
- SSL协商修复(Nathan Mueller)
- 当链接PQconnectDB时，使libpq的 ~/.pgpass 特征工作
- 更新my2pg, ora2pg
- 翻译更新
- 在contrib/lo中添加类型lo和oid之间的转换
- 快速通道代码现在检查调用函数的权限

E.204. 版本 7.3.1

> 发布日期: 2002-12-18

这个版本包含自版本7.3以来的各种补丁。

E.204.1. 迁移到版本 7.3.1

运行 7.3. 版本的用户不需要转储/恢复。 不过，请注意在这个版本里，PostgreSQL的接口库， libpq， 有了一个新的主版本号，在一些情况下需要重新编译客户端代码。

E.204.2. 修改列表

- 当客户端/服务器编码不匹配时修复COPY TO的内核转储(Tom)
- 允许pg_dump在7.2之前的服务器上工作(Philip)
- 修复了contrib/adddepend(Tom)
- 修复删除per-user/per-database配置设置的问题(Tom)
- 修复了contrib/vacuumlo (Tom)
- 当pg_shadow包含MD5口令时允许'password'加密(Bruce)
- 修复了contrib/dbmirror (Steven Singer)
- 修复了优化器(Tom)
- 修复了contrib/tsearch (Teodor Sigaev, Magnus)
- 允许地区名称混合大小写(Nicolai Tufar)
- 增加libpq库的主要版本书数（Bruce)
- 修复了pg_hba.conf错误报告(Bruce, Neil)
- 添加了SCO Openserver 5.0.4作为支持的平台(Bruce)
- 阻止崩溃服务器的EXPLAIN (Tom)
- 修复了SSL (Nathan Mueller)
- 阻止通过ALTER TABLE创建混合字段(Tom)

E.205. 版本 7.3

> 发布日期: 2002-11-27

E.205.1. 概述

主要的变化有：

模式

模式允许用户在独立的命名空间中创建对象，所以两个人或应用可以有相同名字的表。 还有一个为共享表提供的公共模式。表/索引的创建可以通过在公共模式上删除权限来限制。

删除字段

PostgreSQL现在支持 `ALTER TABLE ... DROP COLUMN` 功能。

表函数

返回多个行和/或多个列的函数比以前更好用了。你可以在 `SELECT` `FROM` 子句中调用这样的一个"表函数"， 把它的输出当做一个表。还有PL/pgSQL函数现在可以返回集合了。

预备查询

PostgreSQL现在支持预备查询，以提升性能。

依赖追踪

PostgreSQL现在记录对象依赖关系，允许了在许多方面的改进。 `DROP` 语句现在接受 `CASCADE` 或 `RESTRICT` 来控制是否删除依赖对象。

权限

函数和过程语言现在有权限了，并且函数可以定义为用它们的创建者的权限运行。

国际化

现在总是启用多字节和本地支持。

日志

各种日志选项都增强了。

接口

大量的接口已经移动到了http://gborg.postgresql.org， 在这里它们可以独立的开发和发布。

函数/标识符

缺省的，函数现在可以接受多达32个参数，标识符可以多达63个字节长度。 还有， `OPAQUE` 现在已经废弃了：特定的"pseudo-datatypes" 在函数参数和结果类型中表示 `OPAQUE` 之前的含义。

E.205.2. 迁移到版本 7.3

那些想要从任何先前的版本中迁移数据的需要使用pg_dump的转储/恢复。 如果你的应用检索系统表，那么需要做额外的改变，由于7.3中模式的引入；要获取更多信息， 请参阅http://developer.postgresql.org/~momjian/upgrade_tips_7.3。

观察下列的不一致性：

- 不再支持6.3之前的客户端。
- `pg_hba.conf` 现在有一个用户名和额外特征的字段。 现存的文件需要调整。
- 几个 `postgresql.conf` 日志参数已经重命名了。
- `LIMIT #,#` 已经禁用了，请使用 `LIMIT # OFFSET #` 。
- 带有字段列表的 `INSERT` 语句必须为每个声明的字段声明值。例如， `INSERT INTO tab (col1, col2) VALUES ('val1')` 现在是非法的。 如果 `INSERT` 没有字段列表，那么仍然允许提供少于期望的字段。
- `serial` 字段不再自动 `UNIQUE` ；因此，不会自动创建一个索引。
- 现在在退出的事务内部的 `SET` 命令会回滚。
- `COPY` 不再认为丢失的后续字段为空。需要指定所有的字段。 (不过，可以通过在 `COPY` 命令中声明一个字段列表达到相似的效果。)
- 数据类型 `timestamp` 现在相当于 `timestamp without time zone` ， 而不是 `timestamp with time zone` 。
- 7.3之前的数据库加载到7.3将没有 `serial` 字段、唯一约束和外键的新对象依赖性。 参阅目录 `contrib/adddepend/` 获取一个详细的描述和一个添加这样的依赖性的脚本。
- 不再允许空字符串(`''`)作为整数字段的输入。原先，它被隐式的解释为0。

E.205.3. 修改列表

E.205.3.1. 服务器操作

- 添加pg_locks视图以显示锁(Neil)

- 为密码协商内存分配修复安全性(Neil)
- 删除对版本 0 FE/BE 协议的支持（PostgreSQL 6.2和之前的版本）(Tom)
- 为超级用户保留最后几个后端槽位，参数superuser_reserved_connections控制这个(Nigel J. Andrews)

E.205.3.2. 性能

- 通过一次调用localtime()改善开始菜单(Tom)
- 为快速启动在平面文件中缓存系统目录信息(Tom)
- 改善索引信息的缓存(Tom)
- 优化器改善(Tom, Fernando Nasser)
- 目录缓存现在存储查找失败(Tom)
- 改善哈希函数(Neil)
- 提高查询标记和网络处理的性能(Peter)
- 为大对象存储提高速度(Mario Weilguni)
- 在第一次查询时标记过期的索引项，节省稍后的heap抓取(Tom)
- 避免过多的NULL位图填充(Manfred Koizar)
- 为Solaris提升性能添加BSD监听的qsort() (Bruce)
- 通过4个字节减少每行开销(Manfred Koizar)
- 修复GEQO优化器bug(Neil Conway)
- 使WITHOUT OID实际上保存每行4个字节(Manfred Koizar)
- 添加default_statistics_target变量以声明ANALYZE buckets (Neil)
- 使用本地缓冲区缓存临时表，这样没有WAL开销(Tom)
- 在大表上提升免费空间映射性能(Stephen Marshall, Tom)
- 提升了WAL写的并发性(Tom)

E.205.3.3. 权限

- 在函数和过程语言上添加权限(Peter)

- 添加OWNER到CREATE DATABASE，这样超级用户可以代表非特权用户创建数据库 (Gavin Sherry, Tom)
- 添加新对象权限位EXECUTE 和 USAGE (Tom)
- 添加SET SESSION AUTHORIZATION DEFAULT 和 RESET SESSION AUTHORIZATION (Tom)
- 允许用函数所有者的权限执行函数(Peter)

E.205.3.4. 服务器配置

- 现在服务器日志信息标记为LOG，不是DEBUG (Bruce)
- 添加用户字段到pg_hba.conf (Bruce)
- log_connections在日志文件中输出两行信息(Tom)
- 从postgresql.conf中移除debug_level，现在是server_min_messages (Bruce)
- 为每用户/数据库初始化新建ALTER DATABASE/USER ... SET命令(Peter)
- 新参数server_min_messages 和 client_min_messages控制哪条信息发送给服务器日志和客户端应用(Bruce)
- 允许pg_hba.conf声明逗号隔开的用户/数据库列表，前置+的分组名和前置@的文件名 (Bruce)
- 移除二次密码文件功能和pg_password工具(Bruce)
- 为数据库本地用户名添加变量db_user_namespace (Bruce)
- 改善SSL (Bear Giles)
- 使默认的存储密码加密(Bruce)
- 允许pg_statistics通过调用pg_stat_reset()重置 (Christopher)
- 添加log_duration 参数 (Bruce)
- 将debug_print_query重命名为log_statement (Bruce)
- 将show_query_stats 重命名为 show_statement_stats (Bruce)
- 添加了参数log_min_error_statement到错误时输出到日志的命令(Gavin)

E.205.3.5. 查询

- 使游标不敏感，意味着不改变它们的内容(Tom)

- 禁用 LIMIT #,# 语法；现在只支持 LIMIT # OFFSET # (Bruce)
- 增加标识符长度为63(Neil, Bruce)
- UNION修复了合并不同长度的 >= 3 字段(Tom)
- 添加DEFAULT关键字到 INSERT, e.g., INSERT ... (..., DEFAULT, ...) (Rod)
- 通过使用ALTER COLUMN ... SET DEFAULT允许视图有缺省值(Neil)
- 未能INSERT没有提供所有字段值的字段列表，如：INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
- 修复join别名(Tom)
- 修复了FULL OUTER JOINs (Tom)
- 改善无效标识符和位置的报告(Tom, Gavin)
- 修复 OPEN cursor(args) (Tom)
- 允许'ctid'在视图和currtid(viewname)中使用 (Hiroshi)
- 修复CREATE TABLE AS with UNION (Tom)
- SQL99语法改善(Thomas)
- 添加statement_timeout变量到取消查询(Bruce)
- 允许预备查询PREPARE/EXECUTE (Neil)
- 允许FOR UPDATE出现在LIMIT/OFFSET后面(Bruce)
- 添加变量自动提交(Tom, David Van Wie)

E.205.3.6. 对象操作

- 在 CREATE DATABASE 中使等号可选(Gavin Sherry)
- 使ALTER TABLE OWNER也改变索引的所有者(Neil)
- 新的ALTER TABLE tabname ALTER COLUMN colname SET STORAGE controls TOAST storage，压缩(John Gray)
- 添加模式支持， CREATE/DROP SCHEMA (Tom)
- 为临时表创建模式(Tom)
- 为模式搜索添加变量search_path (Tom)
- 添加了ALTER TABLE SET/DROP NOT NULL (Christopher)

- 新建CREATE FUNCTION波动水平(Tom)
- 使规则名只在每个表中唯一(Tom)
- 添加'ON tablename'子句到DROP RULE 和 COMMENT ON RULE (Tom)
- 添加ALTER TRIGGER RENAME (Joe)
- 新加current_schema() 和 current_schemas()查询功能(Tom)
- 允许函数返回多行（表函数）(Joe)
- 为了一致性，使WITH在CREATE DATABASE中可选(Bruce)
- 添加对象依赖追踪(Rod, Tom)
- 添加RESTRICT/CASCADE到DROP命令 (Rod)
- 为非检查约束添加ALTER TABLE DROP (Rod)
- 自动破坏带有SERIAL的表的DROP时顺序(Rod)
- 如果字段被外键使用那么阻止删除该字段(Rod)
- 当删除了对象时，自动删除约束/函数(Rod)
- 添加了CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- 添加了ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- 阻止继承的字段被删除或重命名(Alvaro Herrera)
- 修复外键约束在中间的数据状态时没有错误(Stephan)
- 传播列或表重命名为外键约束
- 添加CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- 添加CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- 使规则按照字母顺序执行，返回更可预见的值(Tom)
- 触发器现在按照字母顺序触发(Tom)
- 添加/contrib/adddepend以处理7.3之前的对象依赖(Rod)
- 当插入/更新值时允许更好的转换(Tom)

E.205.3.7. 实用命令

- COPY TO输出内嵌回车符，新行为\r 和 \n (Tom)

- 允许COPY FROM中的DELIMITER是8位的(Tatsuo)
- 为了性能，使pg_dump 使用 ALTER TABLE ADD PRIMARY KEY (Neil)
- 在多重语句规则中禁用括号 (Bruce)
- 禁止在一个函数内部调用VACUUM (Bruce)
- 允许dropdb和其他脚本使用带有空格的标识符(Bruce)
- 限制数据库注释更改为当前数据库
- 允许在操作符上注释，不依赖于潜在的函数(Rod)
- 在退出的事务中回滚SET命令(Tom)
- EXPLAIN现在作为查询输出(Tom)
- 显示条件查询和EXPLAIN中排序键(Tom)
- 为单个事务添加'SET LOCAL var = value'以设置配置变量(Tom)
- 允许ANALYZE运行在一个事务中(Bruce)
- 用新的WITH子句改善COPY语法，保持向后兼容性(Bruce)
- 修复pg_dump在非ASCII转储中一致的输出标签(Bruce)
- 使外键约束在转储文件中更清晰(Rod)
- 添加COMMENT ON CONSTRAINT (Rod)
- 允许COPY TO/FROM声明字段名(Brent Verner)
- 将UNIQUE 和 PRIMARY KEY约束作为ALTER TABLE转储(Rod)
- SHOW输出一个查询结果(Joe)
- 生成错误在短的COPY行上而不是在填充的NULL上(Neil)
- 修复CLUSTER保留所有的表属性(Alvaro Herrera)
- 新建pg_settings表以查看/修改GUC设置(Joe)
- 添加智能引用，可移植性提升至pg_dump输出(Peter)
- 作为SERIAL转储出序列字段(Tom)
- 启用大文件支持，pg_dump >2G (Peter, Philip Warner, Bruce)
- 禁止在包含在参考约束中的表上TRUNCATE (Rod)
- 使TRUNCATE也自动截断关系的toast表 (Tom)

- 添加clusterdb实用将自动集群基于先前CLUSTER操作的全部数据库(Alvaro Herrera)
- 彻底检查pg_dumpall (Peter)
- 允许对TOAST表进行REINDEX (Tom)
- 应用START TRANSACTION, 每个 SQL99 (Neil)
- 当页分裂影响容量删除时修复罕见的索引损坏(Tom)
- 为继承修复ALTER TABLE ... ADD COLUMN(Alvaro Herrera)

E.205.3.8. 数据类型和函数

- 修复factorial(0)返回1 (Bruce)
- 改善Date/time/timezone (Thomas)
- 修复数组切片提取(Tom)
- 修复extract/date_part为时间戳报告适当的微秒(Tatsuo)
- 允许text_substr() 和 bytea_substr()更有效的读取TOAST值(John Gray)
- 添加域支持(Rod)
- 使WITHOUT TIME ZONE为TIMESTAMP和TIME数据类型的缺省(Thomas)
- 在配置中使用--enable-integer-datetimes允许64位整数交替日期/时间类型存储模式(Thomas)
- 使timezone(timestamptz)返回时间戳而不是一个字符串(Thomas)
- 时间的日期/时间类型中允许小数秒在1BC之前(Thomas)
- 限制时间戳数据类型精度为6个小数位(Thomas)
- 更改时区转换函数timetz()为timezone() (Thomas)
- 添加配置变量数据类型和时区(Tom)
- 添加OVERLAY(), 允许替换字符串的子串(Thomas)
- 添加SIMILAR TO (Thomas, Tom)
- 添加正规表达式SUBSTRING(string FROM pat FOR escape) (Thomas)
- 添加LOCALTIME 和 LOCALTIMESTAMP 函数 (Thomas)
- 使用CREATE TYPE typename AS (column)添加命名的复合类型(Joe)

- 允许在表别名子句中定义复合类型(Joe)
- 添加新的API以简化C语言表函数的创建(Joe)
- 从对SQL99函数的调用中删除ODBC兼容的空括号，因为这些括号不匹配标准(Thomas)
- 允许macaddr数据类型接受12个带有分隔符的十六进制数字(Mike Wyer)
- 添加CREATE/DROP CAST (Peter)
- 添加IS DISTINCT FROM 操作 (Thomas)
- 添加SQL99 TREAT()函数，CAST()的同义词(Thomas)
- 添加pg_backend_pid()输出后端pid (Bruce)
- 添加IS OF / IS NOT OF类型谓词 (Thomas)
- 允许位字符串常量不是完全指定的长度(Thomas)
- 允许在8字节整数和位字符串之间转换(Thomas)
- 实现十六进制文字转换为位字符串文字(Thomas)
- 允许表函数显示在FROM子句中(Joe)
- 增加函数参数的最大数量为32 (Bruce)
- 不再为SERIAL字段自动创建索引(Tom)
- 添加current_database() (Rod)
- 修复cash_words()不溢出缓存(Tom)
- 添加函数replace(), split_part(), to_hex() (Joe)
- 为bytea修复LIKE为右参数(Joe)
- 阻止SELECT cash_out(2)导致的崩溃(Tom)
- 修复to_char(1,'FM999.99')返回一个时期(Karel)
- 修复返回OPAQUE的触发器/类型/语言函数返回合适的类型(Tom)

E.205.3.9. 国际化

- 添加额外的编码: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- 缺省启用本地支持(Peter)

- 添加语言环境变量(Peter)
- 在PQescapeBytea/PQunescapeBytea中为多字节逃逸byes >= 0x7f (Tatsuo)
- 添加语言环境意识到规则表达式字符类
- 缺省启用多字节支持(Tatsuo)
- 添加GB18030多字节支持(Bill Huang)
- 添加CREATE/DROP CONVERSION，允许可加载的编码(Tatsuo, Kaori)
- 添加pg_conversion表(Tatsuo)
- 添加SQL99 CONVERT()函数(Tatsuo)
- pg_dumpall, pg_controldata, 和 pg_resetxlog现在国家语言意识(Peter)
- 新的和更新的翻译

E.205.3.10. 服务器端语言

- 允许递归的SQL函数(Peter)
- 更改PL/Tcl构造以使用配置的编译器和Makefile.shlib (Peter)
- 彻底检查PL/pgSQL FOUND变量，使其更加Oracle兼容(Neil, Tom)
- 允许PL/pgSQL处理引用的标识符(Tom)
- 允许设置返回PL/pgSQL函数 (Neil)
- 使PL/pgSQL意识到模式(Joe)
- 删除一些内存溢出(Nigel J. Andrews, Tom)

E.205.3.11. psql

- 为了与7.2.0兼容，不要小写psql \connect数据库名(Tom)
- 添加psql \timing到用户查询时间(Greg Sabino Mullane)
- 使psql \d显示索引信息(Greg Sabino Mullane)
- 新建psql \dD显示域(Jonathan Eisler)
- 允许psql在视图上显示规则(Paul ?)
- 修复psql变量替换(Tom)
- 允许psql \d显示临时表结构(Tom)

- 允许psql \d显示外键(Rod)
- 修复\?以纪念\pset pager (Bruce)
- 使psql在启动时报告它的版本号(Tom)
- 允许\copy指定字段名(Tom)

E.205.3.12. libpq

- 添加~/.pgpass存储主机/用户密码组合(Alvaro Herrera)
- 添加PQunescapeBytea()函数到libpq (Patrick Welche)
- 修复在非阻塞连接上发送大查询(Bernhard Herzog)
- 修复libpq在Win9X上使用定时器(David Ford)
- 允许libpq通知用不同长度的标识符处理服务器(Tom)
- 添加libpq PQescapeString() 和 PQescapeBytea()到Windows (Bruce)
- 用非阻塞连接修复SSL (Jack Bates)
- 添加libpq连接超时参数(Denis A Ustimenko)

E.205.3.13. JDBC

- 允许JDBC用JDK 1.4编译(Dave)
- 添加JDBC 3支持(Barry)
- 允许JDBC通过添加?loglevel=X到连接的URL来设置日志级别(Barry)
- 添加Driver.info()信息，输出版本号(Barry)
- 添加可更新的结果集(Raghu Nidagal, Dave)
- 添加对可调用语句的支持(Paul Bethe)
- 添加查询取消能力
- 添加立即刷新(Dave)
- 修复MD5加密处理多字节服务器(Jun Kawai)
- 添加对预备语句的支持(Barry)

E.205.3.14. 各种接口

- 修复ECPG bug，关于单引号中的八进制数字(Michael)
- 引动src/interfaces/libpgeasy到http://gborg.postgresql.org (Marc, Bruce)
- 改善Python接口(Elliot Lee, Andrew Johnson, Greg Copeland)
- 添加libpgtcl连接关闭事件(Gerhard Hintermayer)
- 移动src/interfaces/libpq++到http://gborg.postgresql.org (Marc, Bruce)
- 移动src/interfaces/odbc到http://gborg.postgresql.org (Marc)
- 移动src/interfaces/libpgeasy到http://gborg.postgresql.org (Marc, Bruce)
- 移动src/interfaces/perl5到http://gborg.postgresql.org (Marc, Bruce)
- 从主干上删除src/bin/pgaccess，现在在http://www.pgaccess.org (Bruce)
- 添加pg_on_connection_loss命令到libpgtcl (Gerhard Hintermayer, Tom)

E.205.3.15. 源码

- 修复并行进行(Peter)
- AIX修复了连接Tcl (Andreas Zeugswetter)
- 允许PL/Perl在Cygwin下建立(Jason Tishler)
- 改善MIPS编译(Peter, Oliver Elphick)
- 需要Autoconf版本2.53 (Peter)
- 需要在配置中缺省有readline 和 zlib (Peter)
- 为了性能，允许Solaris使用Intimate Shared Memory (ISM) (Scott Brunza, P.J. Josh Rovero)
- 在编译时总是启用系统日志，删除--enable-syslog选项(Tatsuo)
- 在编译时总是启用多字节，删除--enable-multibyte选项(Tatsuo)
- 在编译时总是启用区域设置，删除--enable-locale选项(Peter)
- 修复Win9x DLL创建(Magnus Naeslund)
- 通过WAL代码在Windows、BeOS上修复link() (Jason Tishler)
- 添加sys/types.h 到 c.h，并从主文件中删除(Peter, Bruce)
- 修复AIX挂在SMP机器上(Tomoyuki Niijima)

- AIX SMP 挂修复(Tomoyuki Niijima)
- 修复1970以前的日期在新的glibc库上处理(Tom)
- 修复PowerPC SMP锁定(Tom)
- 阻止使用gcc -ffast-math (Peter, Tom)
- Bison >= 1.50现在需要开发者建立
- Kerberos 5 支持现在创建时带有Heimdal (Peter)
- 在列出SQL特性的用户手册中添加附录(Thomas)
- 改善可加载的模块连接到使用RTLD_NOW (Tom)
- 新的错误级别WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- 新建src/port目录保存替换的libc函数(Peter, Bruce)
- 为模式新建pg_namespace系统目录(Tom)
- 为模式添加pg_class.relnamespace(Tom)
- 为模式添加pg_type.typnamespace (Tom)
- 为模式添加pg_proc.pronamespace (Tom)
- 调整聚合有pg_proc条目(Tom)
- 系统关系现在有了它们自己的命名空间，不再需要pg_* test (Fernando Nasser)
- 重命名TOAST索引为_*index*而不是_idx (Neil)
- 为操作符、操作符类添加命名空间(Tom)
- 添加额外的检查到服务器控制文件(Thomas)
- 新增Polish FAQ (Marcin Mazurek)
- 添加Posix信号灯支持(Tom)
- 文档需要重建索引(Bruce)
- 重命名一些内部的标识符以简化Windows编译(Jan, Katherine Ward)
- 添加计算磁盘空间的文件(Bruce)
- 从GUC中删除KSQO(Bruce)
- 修复rtree中的内存溢出(Kenneth Been)
- 为一致性修复一些错误消息(Bruce)

- 删除未使用的系统表字段(Peter)
- 在适当的地方使系统字段为NOT NULL (Tom)
- 为了支持snprintf()清除snprintf的使用(Neil, Jukka Holappa)
- 删除OPAQUE创建特定的子类型(Tom)
- 在数组内部处理清理(Joe, Tom)
- 不允许pg_atoi('') (Bruce)
- 删除参数wal_files因为WAL文件现在回收利用了(Bruce)
- 添加版本号到heap页(Tom)

E.205.3.16. 贡献包

- /contrib/array中允许inet数组(Neil)
- GiST 修复 (Teodor Sigaev, Neil)
- 升级/contrib/mysql
- 添加/contrib/dbsize，它显示了没有vacuum的表大小(Peter)
- 添加/contrib/intagg，整数聚合器例程(mlw)
- 改善/contrib/oid2name (Neil, Bruce)
- 改善/contrib/tsearch (Oleg, Teodor Sigaev)
- 清理/contrib/rserver (Alexey V. Borzov)
- 更新/contrib/oracle转换实用程序(Gilles Darold)
- 更新/contrib/dblink (Joe)
- 改善/contrib/vacuumlo支持的选项(Mario Weilguni)
- 改善/contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- 添加/contrib/reindexdb实用程序(Shaun Thomas)
- 添加索引到/contrib/isbn_issn (Dan Weston)
- 添加/contrib/dbmirror (Steven Singer)
- 改善/contrib/pgbench (Neil)
- 添加/contrib/tablefunc表函数示例(Joe)

- 为树形结构添加/contrib/ltree数据类型(Teodor Sigaev, Oleg Bartunov)
- 移动/contrib/pg_controldata, pg_resetxlog到主树里(Bruce)
- 修复/contrib/cube (Bruno Wolff)
- 改善/contrib/fulltextindex (Christopher)

E.206. 版本 7.2.8

> 发布日期: 2005-05-09

这个版本包含自7.2.7以来的各种补丁，包括一个安全相关的问题。

E.206.1. 迁移到版本 7.2.8

运行7.2.X的用户不需要转储/恢复。

E.206.2. 修改列表

- 修复过时的竞态条件，该条件允许一个事务因为某些目的被视作已提交(如SELECT FOR UPDATE)

  这是一个非常严重的bug，因为它会导致表面上数据的不一致被应用短暂的看到。

- 修复关系扩展和VACUUM之间的竞态条件。

  理论上这可能导致丢失新插入的一页数据，尽管该情况看起来概率很低。 没有导致多于一个维护失败的已知情况。

- 为 `TIME WITH TIME ZONE` 值修复 `EXTRACT(EPOCH)`
- 在plpgsql中额外的缓存溢出检查(Neil)
- 修复pg_dump以正确的转储包含 `%` 的索引名和触发器名
- 阻止 `to_char(interval)` 转储月份相关格式的内核
- 为更新的OpenSSL构建修复 `contrib/pgcrypto` (Marko Kreen)

E.207. 版本 7.2.7

> 发布日期: 2005-01-31

这个版本包含自7.2.6以来的各种补丁，包括几个安全相关的问题。

E.207.1. 迁移到版本 7.2.7

运行7.2.X的用户不需要转储/恢复。

E.207.2. 修改列表

- 不允许非超级用户 `LOAD`

  在平台上会自动执行共享库的初始化函数（至少包括Windows和ELF-based Unixen），`LOAD` 可以用来使服务器执行任意的代码。感谢NGS Software报告这个问题。

- 添加需要的STRICT标记到一些贡献包功能(Kris Jurka)
- 当plpgsql的游标声明有太多的参数时避免缓冲区溢出(Neil)
- 修复全外连接和右外连接的规划错误

  连接的结果错误的与左输入有相同的排序。这不只是发送错误排序的输出给用户， 还在嵌套的合并连接的情况下给出完全错误的回复。

- 修复SQL和GERMAN数据类型中负数间隔的显示

E.208. 版本 7.2.6

> 发布日期: 2004-10-22

这个版本包含各种自7.2.5以来的修复。

E.208.1. 迁移到版本 7.2.6

运行7.2.X版本的用户不需要转储/恢复。

E.208.2. 修改列表

- 修复可能未能更新到磁盘上的错误

  在极少数情况下，这个疏忽可能会导致"不能访问事务状态"错误， 这是一个潜在的数据丢失bug。

- 确保散列的外连接不丢失元组

  使用哈希连接规划的非常大的左连接可能没有输出不匹配的左侧行，只给出右侧数据分布。

- 不允许作为root运行pg_ctl

  这是为了防范任何可能的安全问题。

- 避免在/tmp中的make_oidjoins_check中使用临时文件

  这已经作为一个安全问题上报，尽管它几乎不值得关心，因为非开发者任何情况下都没有理由使用这个脚本。

- 更新到更新的Bison版本

E.209. 版本 7.2.5

> 发布日期: 2004-08-16

这个版本包含自7.2.4以来的各种修复。

E.209.1. 迁移到版本 7.2.5

运行7.2.X的用户不需要转储/恢复。

E.209.2. 修改列表

- 阻止崩溃期间可能的已提交事务的丢失。

  由于事务提交和检查点之间连锁不足，正好在最近的检查点之前提交的事务可能会丢失， 全部丢失或部分丢失，紧跟着数据库崩溃并重启。这是一个严重的bug， 自PostgreSQL 7.1就已经存在了。

- 修复btree搜索和第一个root页面分裂并行的极端情况
- 修复 `to_ascii` 里的缓冲区溢出(Guido Notari)
- 修复机器上无符号字符的死锁检测里的内核转储
- 修复运行Async_NotifyHandler之后未能响应 `pg_ctl stop -m fast`
- 修复pg_dump中的内存泄露
- 避免与 `isblank()` 函数或宏的系统定义冲突

E.210. 版本 7.2.4

> 发布日期: 2003-01-30

这个版本包含各种自7.2.3以来的修复，包括阻止可能的数据丢失的修复。

E.210.1. 迁移到版本 7.2.4

运行7.2.*版本的用户不需要转储/恢复。

E.210.2. 修改列表

- 修复一些VACUUM "No one parent tuple was found"错误的额外情况
- 阻止VACUUM在一个函数内部调用 (Bruce)
- 确保pg_clog更新在标记检查点完成之前同步到磁盘
- 避免在大的哈希连接期间整数溢出
- 当pg_group.grolist足够大时使GROUP命令工作
- 修复日期时间表中的错误；一些时区名称不被认可
- 修复circle_poly(), path_encode(), path_add()中的整数溢出 (Neil)
- 修复lseg_eq(), lseg_ne(), lseg_center()中长期存在的逻辑错误

E.211. 版本 7.2.3

> 发布日期: 2002-10-01

这个版本包含各种自版本7.2.2以来的修复，包括阻止可能的数据丢失的修复。

E.211.1. 迁移到版本 7.2.3

运行7.2.*版本的用户不需要转储/恢复。

E.211.2. 修改列表

- 阻止可能的压缩事务日志的丢失 (Tom)
- 阻止非超级用户增加最近的vacuum信息 (Tom)
- 在最新版本的glibc中处理1970以前的数据值 (Tom)
- 修复服务器关闭期间可能的中止
- 阻止自旋锁在SMP PPC机器上不挂断 (Tomoyuki Niijima)
- 修复pg_dump以适当的转储FULL JOIN USING (Tom)

E.212. 版本 7.2.2

> 发布日期: 2002-08-23

这个版本包含各种自版本7.2.1以来的修复。

E.212.1. 迁移到版本 7.2.2

运行7.2.*版本的用户不需要转储/恢复。

E.212.2. 修改列表

- 允许在PL/pgSQL中执行"CREATE TABLE AS ... SELECT" (Tom)
- 修复压缩事务日志id概括 (Tom)
- 修复PQescapeBytea/PQunescapeBytea以便它们处理字节 > 0x7f (Tatsuo)
- 修复被不存在的长选项调用时的psql和pg_dump崩溃 (Tatsuo)
- 修复调用几何运算符时的死机 (Tom)
- 允许OPEN游标（参数） (Tom)
- 修复rtree_gist索引建立 (Teodor)
- 修复转储用户定义的聚集 (Tom)
- 修复contrib/intarray (Oleg)
- 修复使用圆括号的复杂的UNION/EXCEPT/INTERSECT查询 (Tom)
- 修复pg_convert (Tatsuo)
- 修复长DATA字符串的崩溃 (Thomas, Neil)
- 修复repeat(), lpad(), rpad()和长字符串 (Neil)

E.213. 版本 7.2.1

> 发布日期: 2002-03-21

这个版本包含对版本7.2的各种修复。

E.213.1. 迁移到版本 7.2.1

运行版本7.2的用户不需要转储/恢复。

E.213.2. 修改列表

- 确保序列计数器在崩溃后不要倒退 (Tom)
- 修复pgaccess kanji-conversion关键字捆绑 (Tatsuo)
- 改善优化器 (Tom)
- 改善Cash I/O (Tom)
- 新的俄罗斯常见问题解答
- 编译修复丢失AuthBlockSig (Heiko)
- 增加时区和时区修复 (Thomas)
- 允许psql \connect处理混合情况的数据库和用户名 (Tom)
- 在命令结束时返回适当的OID，即使是ON INSERT规则也一样 (Tom)
- 允许COPY FROM使用8位的DELIMITERS (Tatsuo)
- 为毫秒/微秒修复extract/date_part里的bug (Tatsuo)
- 改善不同长度的多重UNION的处理 (Tom)
- 改善contrib/btree_gist (Teodor Sigaev)
- 改善contrib/tsearch字典，参阅README.tsearch获取附加的安装步骤 (Thomas T. Thai, Teodor Sigaev)
- 修复数组下标的处理 (Tom)
- 允许在PL/pgSQL中执行"CREATE TABLE AS ... SELECT" (Tom)

E.214. 版本 7.2

> 发布日期: 2002-02-04

E.214.1. 概述

这个版本改善了PostgreSQL在高负荷环境下的能力。

这个版本里的主要变化:

VACUUM

Vacuum 不再锁住表，因此就允许普通用户在 vacuum 的时候访问。 新的 `VACUUM FULL` 命令做的就是老的 vacuum，它先锁定该表然后缩小表的磁盘文件.

事务

不再有超过四十亿次事务的安装问题了。

OIDs

OID 现在是可选的。用户现在可以不用带 OID 创建表，以免过多地使用 OID。

优化器

系统现在在 `ANALYZE` 的时候计算直方图列统计，这样就允许选择更好的优化器。

安全性

新的 MD5 加密选项允许我们有更安全的口令存储和传输。 新的 Unix 域套接字认证选项可以在 Linux 和 BSD 系统上使用。

统计

管理员现在可以使用新的表访问统计模块获取有关表和索引的使用方面的更细致的信息.

国际化

程序和库消息现在可以以好几种语言显示。

E.214.2. 迁移到版本 7.2

对于使用任何以前版本的人来说，用 `pg_dump` 进行一次转储/恢复是必需的。

观察下面的不兼容性:

- 这个版本里的 `VACUUM` 命令的语义发生了变化。你可能需要相应更新你的维护程序。
- 在这个版本里，用 `= NULL` 进行的比较将总是返回假(或者更准确地说是 NULL)。 以前的版本自动把这个语法转换成 `IS NULL` 。 你可以使用 `postgresql.conf` 参数重新打开原来的行为。
- `pg_hba.conf` 和 `pg_ident.conf` 配置现在只有在收到一个SIGHUP信号之后才重新装载，而不是每次连接就重新装载。
- 函数 `octet_length()` 现在返回未压缩的数据长度。
- 日期/时间数值 `'current'` 不再可用。这方面你需要改写你的应用.
- `timestamp()` , `time()` , 和 `interval()` 函数不可再用。应该用 `timestamp 'string'` 或 `CAST` 代替 `timestamp()` 。

`SELECT ... LIMIT #,#` 语法将在下一个版本中删除。 你应该改写你的查询，利用 LIMIT 和 OFFSET 子句，也就是说 `LIMIT 10 OFFSET 20` 。

E.214.3. 修改列表

E.214.3.1. 服务器操作

- 在一个单独的路径中创建临时文件 (Bruce)
- 在主进程启动时删除孤立的临时文件 (Bruce)
- 添加唯一索引到一些系统表 (Tom)
- 系统表操作符重组 (Oleg Bartunov, Teodor Sigaev, Tom)
- 重命名pg_log to pg_clog (Tom)
- 启用SIGTERM, SIGQUIT以杀死后端 (Jan)
- 在许多后端上删除编译时间限制 (Tom)
- 更好的清理信号灯资源失败 (Tatsuo, Tom)
- 允许安全事务ID概括 (Tom)
- 从一些系统表中删除OID (Tom)
- 删除"triggered data change violation"错误校验 (Tom)
- 预备/保存规划的SPI入口创建 (Jan)
- 允许SPI字段函数为系统字段工作 (Tom)

- 改善长值压缩 (Tom)
- 统计收集表、索引访问 (Jan)
- 截断超长的序列名为合理的值 (Tom)
- 以毫秒测量事务时间 (Thomas)
- 修复TID顺序扫描 (Hiroshi)
- 超级用户ID现在固定在1 (Peter E)
- 新增pg_ctl "reload"选项 (Tom)

E.214.3.2. 性能

- 改善优化器 (Tom)
- 为优化器新增直方图列统计 (Tom)
- 重新使用预写式日志文件而不是丢弃它们 (Tom)
- 改善缓存 (Tom)
- IS NULL, IS NOT NULL优化器改善 (Tom)
- 改善锁管理器以减少锁的争用 (Tom)
- 为索引访问支持函数保持relcache条目 (Tom)
- 允许NUMERIC中NaN和无穷的更好的选择性 (Tom)
- R-tree性能改善 (Kenneth Been)
- B-tree分离更高效 (Tom)

E.214.3.3. 权限

- 更改UPDATE, DELETE权限为不同的 (Peter E)
- 新增REFERENCES, TRIGGER 权限 (Peter E)
- 允许一次多于一个用户GRANT/REVOKE to/from (Peter E)
- 新增has_table_privilege()函数 (Joe Conway)
- 允许非超级用户vacuum数据库 (Tom)
- 新增SET SESSION AUTHORIZATION命令 (Peter E)
- 修复在新创建的表上修改权限的bug (Tom)

- 不允许非超级用户访问pg_statistic，添加用户可访问的视图 (Tom)

E.214.3.4. 客户端认证

- 在做认证以阻止挂起之前分叉主进程 (Peter E)
- 在Linux, *BSD平台的Unix域套接字上添加标识符认证 (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- 添加使用MD5加密的口令认证方法 (Bruce)
- 允许存储的口令使用MD5加密 (Bruce)
- PAM 认证 (Dominic J. Eidson)
- 只在启动和SIGHUP时加载pg_hba.conf 和 pg_ident.conf (Bruce)

E.214.3.5. 服务器配置

- 现在在运行时可设定一些时区缩写的解释为澳大利亚而不是北美 (Bruce)
- 新增参数设置缺省的事务隔离级别 (Peter E)
- 新增参数启用"expr = NULL" 到 "expr IS NULL"的转换，缺省为off (Peter E)
- 新增参数控制VACUUM的内存使用 (Tom)
- 新增参数设置客户端认证超时 (Tom)
- 新增参数设置打开文件的最大数量 (Tom)

E.214.3.6. 查询

- INSERT规则添加的声明现在在INSERT之后执行 (Jan)
- 阻止在目标列表中的纯粹的关系名 (Bruce)
- NULL现在在ORDER BY中在所有的正常值之后排序 (Tom)
- 新增IS UNKNOWN, IS NOT UNKNOWN布尔测试 (Tom)
- 新增SHARE UPDATE EXCLUSIVE锁模式 (Tom)
- 新增EXPLAIN ANALYZE命令，显示运行时间和行计数 (Martijn van Oosterhout)
- 修复LIMIT和子查询的问题 (Tom)
- 修复LIMIT, DISTINCT ON推入子查询 (Tom)

- 修复嵌套的EXCEPT/INTERSECT (Tom)

E.214.3.7. 模式操作

- 修复临时表中的SERIAL (Bruce)
- 允许临时序列 (Bruce)
- 序列现在内部使用int8 (Tom)
- 新增SERIAL8创建带有序列的int8字段，缺省仍然为SERIAL4 (Tom)
- 使用WITHOUT OIDS使OID可选 (Tom)
- 添加%TYPE语法到CREATE TYPE (Ian Lance Taylor)
- 为CHECK约束添加ALTER TABLE / DROP CONSTRAINT (Christopher Kings-Lynne)
- 新增CREATE OR REPLACE FUNCTION以改变现有的函数(保留函数 OID) (Gavin Sherry)
- 添加ALTER TABLE / ADD [UNIQUE | PRIMARY] (Christopher Kings-Lynne)
- 允许在视图中重命名字段
- 使ALTER TABLE / RENAME COLUMN更新索引的字段名 (Brent Verner)
- 修复继承的表的ALTER TABLE / ADD CONSTRAINT ... CHECK (Stephan Szabo)
- ALTER TABLE RENAME正确的更新外键触发器参数 (Brent Verner)
- DROP AGGREGATE 和 COMMENT ON AGGREGATE现在接受一个aggtype (Tom)
- 为SQL函数添加自动返回类型数据转换 (Tom)
- 允许GiST索引处理NULL和多键的索引 (Oleg Bartunov, Teodor Sigaev, Tom)
- 启用部分索引 (Martijn van Oosterhout)

E.214.3.8. 工具命令

- 添加RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP现在允许选项顺序任意 (Vince)
- 添加LOCK A, B, C功能 (Neil Padgett)
- 新增ENCRYPTED/UNENCRYPTED选项到CREATE/ALTER USER (Bruce)
- 新增轻型VACUUM不锁定表；老的语义作为VACUUM FULL可用 (Tom)

- 在视图上禁用COPY TO/FROM (Bruce)
- COPY DELIMITERS字符串必须恰好是一个字符 (Tom)
- VACUUM关于索引元组少于堆的警告现在只在合适的时候出现 (Martijn van Oosterhout)
- 为CREATE INDEX修复权限检查 (Tom)
- 不允许不适当的使用CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

E.214.3.9. 数据类型和函数

- SUM(), AVG(), COUNT()现在为了速度在内部使用int8 (Tom)
- 添加convert(), convert2() (Tatsuo)
- 新增函数bit_length() (Peter E)
- 使"n" 在 CHAR(n)/VARCHAR(n)中代表字母而不是字节 (Tatsuo)
- CHAR(), VARCHAR()现在拒绝太长的字符串 (Peter E)
- BIT VARYING现在拒绝太长的位字符串 (Peter E)
- BIT现在拒绝不匹配声明的尺寸的位字符串 (Peter E)
- INET, CIDR文本转换功能 (Alex Pilosov)
- INET, CIDR操作符 << 和 <<= 可索引 (Alex Pilosov)
- Bytea ###现在需要有效的三位八进制数字
- Bytea比较改善，现在支持 =, <>, >, >=, <, 和 <=
- Bytea现在支持B-tree索引
- Bytea现在支持LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea现在支持串联
- 新增bytea函数：position, substring, trim, btrim, 和 length
- 新增encode()函数模式，"escaped"，转换最低限度逃逸的bytea to/from文本
- 添加pg_database_encoding_max_length() (Tatsuo)
- 添加pg_client_encoding() 函数 (Tatsuo)
- now()返回毫秒精度的时间 (Thomas)
- 新增 TIMESTAMP WITHOUT TIMEZONE 数据类型 (Thomas)

- 添加ISO日期/时间规格："T", yyyy-mm-ddThh:mm:ss (Thomas)
- 新增 xid/int 比较函数 (Hiroshi)
- 添加精度到TIME, TIMESTAMP, 和 INTERVAL数据类型 (Thomas)
- 修改类型强制逻辑为尝试二进制兼容函数优先 (Tom)
- 新增缺省安装的encode()函数 (Marko Kreen)
- 改进 to_*() 转换函数 (Karel Zak)
- 当使用单字节编码时优化LIKE/ILIKE (Tatsuo)
- 在contrib/pgcrypto中新增函数： crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- 改正 translate() 函数的描述 (Bruce)
- 添加INTERVAL参数以SET TIME ZONE (Thomas)
- 添加INTERVAL YEAR TO MONTH (等等)语法 (Thomas)
- 当使用单字节编码时优化长度函数 (Tatsuo)
- 修复path_inter, path_distance, path_length, dist_ppath以处理闭合路径 (Curtis Barrett, Tom)
- octet_length(text)现在返回非压缩的长度 (Tatsuo, Bruce)
- 在日期/时间字面值中处理"July"全名 (Greg Sabino Mullane)
- 一些datatype()函数调用现在的评估不同
- 添加Julian 和 ISO时间声明支持 (Thomas)

E.214.3.10. 国际化

- 在psql, pg_dump, libpq, 和 server中支持国家语言 (Peter E)
- 用汉语（简化了的，传统的）、捷克语、法语、德语、匈牙利语、俄语、瑞典语翻译消息 (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- 使trim, ltrim, rtrim, btrim, lpad, rpad意识到翻译多字节 (Tatsuo)
- 添加LATIN5,6,7,8,9,10支持 (Tatsuo)
- 添加ISO 8859-5,6,7,8支持 (Tatsuo)
- 改正LATIN5意为ISO-8859-9, 而不是 ISO-8859-5 (Tatsuo)
- 使mic2ascii()没有ASCII意识 (Tatsuo)

- 拒绝无效的多字节字符序列 (Tatsuo)

E.214.3.11. PL/pgSQL

- 现在使用SELECT循环入口，允许大的结果集 (Jan)
- 支持CURSOR 和 REFCURSOR (Jan)
- 现在可以返回开放的游标 (Jan)
- 添加ELSEIF (Klaus Reger)
- 改善PL/pgSQL错误报告，包括错误的位置 (Tom)
- 为了兼容性，在游标声明中允许IS或FOR关键字 (Bruce)
- 修复SELECT ... FOR UPDATE (Tom)
- 修复PERFORM返回多行 (Tom)
- 使PL/pgSQL强制使用服务器的类型转换代码 (Tom)
- 内存泄露修复 (Jan, Tom)
- 使尾随的分号可选 (Tom)

E.214.3.12. PL/Perl

- 新增不信任的PL/Perl (Alex Pilosov)
- PL/Perl现在在一些平台上建立，即使libperl不是共享的 (Peter E)

E.214.3.13. PL/Tcl

- 现在报告errorInfo (Vsevolod Lobko)
- 添加spi_lastoid函数 (bob@redivi.com)

E.214.3.14. PL/Python

- ...是新增的 (Andrew Bosma)

E.214.3.15. psql

- \d 显示不重复的索引，主要分组 (Christopher Kings-Lynne)
- 在反斜杠命令里允许尾随的分号 (Greg Sabino Mullane)

- 如果可以，从/dev/tty中读取口令
- 当改变用户和数据库时强制新的口令提示 (Tatsuo, Tom)
- 格式化正确的字段编号为Unicode

E.214.3.16. libpq

- 新增函数PQescapeString()以逃逸命令字符串中的引号 (Florian Weimer)
- 新增函数PQescapeBytea()为使用SQL字符串文本逃逸二进制字符串

E.214.3.17. JDBC

- 返回INSERT的OID (Ken K)
- 处理更多数据类型 (Ken K)
- 在字符串中处理单引号和新行 (Ken K)
- 处理NULL变量 (Ken K)
- 修复时区处理 (Barry Lind)
- 改善Druid支持
- 允许带有非多字节服务器的八位字符 (Barry Lind)
- 支持BIT, BINARY类型 (Ned Wolpert)
- 减少内存使用 (Michael Stephens, Dave Cramer)
- 更新DatabaseMetaData (Peter E)
- 添加DatabaseMetaData.getCatalogs() (Peter E)
- 编码修复 (Anders Bengtsson)
- Get/setCatalog方法 (Jason Davies)
- DatabaseMetaData.getColumns()现在返回列的默认值 (Jason Davies)
- 改善DatabaseMetaData.getColumns()性能 (Jeroen van Vianen)
- 一些JDBC1 和 JDBC2合并了 (Anders Bengtsson)
- 事务性能改进 (Barry Lind)
- 数组修复 (Greg Zoller)
- 序列化添加

- 修复了批处理 (Rene Pijlman)
- ExecSQL方法重组 (Anders Bengtsson)
- 修复了GetColumn() (Jeroen van Vianen)
- 修复了isWriteable()函数 (Rene Pijlman)
- 改善JDBC2一致性测试的通道 (Rene Pijlman)
- 添加bytea类型兼容 (Barry Lind)
- 添加isNullable() (Rene Pijlman)
- JDBC日期/时间测试套件修复 (Liam Stewart)
- 修复SELECT 'id' AS xxx FROM table (Dave Cramer)
- 修复DatabaseMetaData以恰当的显示精度 (Mark Lillywhite)
- 新增getImported/getExported键 (Jason Davies)
- 支持MD5口令加密 (Jeremy Wohl)
- 修复实际使用类型缓存 (Ned Wolpert)

E.214.3.18. ODBC

- 删除查询大小限制 (Hiroshi)
- 删除文本字段大小限制 (Hiroshi)
- 修复多字节模式中的SQLPrimaryKeys (Hiroshi)
- 允许ODBC程序调用 (Hiroshi)
- 改善布尔处理 (Aidan Mountford)
- 大部分配置选项现在可以通过DSN设置 (Hiroshi)
- 多字节，性能修复 (Hiroshi)
- 允许iODBC 或 unixODBC使用驱动程序 (Peter E)
- 支持MD5口令加密 (Bruce)
- 添加更多兼容函数到odbc.sql (Peter E)

E.214.3.19. ECPG

- 应用了EXECUTE ... INTO (Christof Petig)

- 多重描述符支持 (e.g. CARDINALITY) (Christof Petig)
- 修复GRANT参数 (Lee Kindness)
- 修复INITIALLY DEFERRED bug
- 各种bug修复 (Michael, Christof Petig)
- 自动分配指示器变量数组 (int *ind_p=NULL)
- 自动分配字符串数组 (char **foo_pp=NULL)
- 修复ECPGfree_auto_mem
- 所有带有外部链接的函数名现在都有ECPG前缀
- 修复结构的数组 (Michael)

E.214.3.20. 混合接口

- Python 修复 fetchone() (Gerhard Haring)
- 在 Tcl 中合适的地方使用 UTF, Unicode (Vsevolod Lobko, Reinhard Max)
- 添加Tcl COPY TO/FROM (ljb)
- 阻止pg_dump中缺省索引 op 类的输出 (Tom)
- 修复 libpgeasy 内存泄露 (Bruce)

E.214.3.21. 建立和安装

- 配置、动态加载和共享库的修复 (Peter E)
- 修复了 QNX 的四个端口 (Bernd Tegge)
- 修复了 Cygwin 和 Windows 端口 (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- 修复了 Windows 接口通讯失败 (Magnus, Mikhail Terekhov)
- 硬编译修复 (Oliver Elphick)
- BeOS修复 (Cyril Velter)
- 删除 configure --enable-unicode-conversion，现在由多字节启用 (Tatsuo)
- AIX 修复 (Tatsuo, Andreas)
- 修复并行进行 (Peter E)

- 安装 SQL 语言手册页到 OS 指定的目录 (Peter E)
- 重命名 config.h 为 pg_config.h (Peter E)
- 整理头文件的安装布局 (Peter E)

E.214.3.22. 源代码

- 删除 SEP_CHAR (Bruce)
- 新增 GUC hooks (Tom)
- 合并 GUC 和命令行处理 (Marko Kreen)
- 删除 EXTEND INDEX (Martijn van Oosterhout, Tom)
- 新增 pgjindent 到 java 代码缩进 (Bruce)
- 在 C++ 下编译时删除 true/false 的定义 (Leandro Fanzone, Tom)
- pgindent 修复 (Bruce, Tom)
- 在适当的地方用 strcmp() 替换 strcasecmp() (Peter E)
- 改善 Dynahash 可移植性 (Tom)
- 在自旋锁结构中添加 'volatile' 用法
- 改善信号处理逻辑 (Tom)

E.214.3.23. 贡献包

- 新增 contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- 新增 contrib/tsearch 全文本索引 (Oleg, Teodor Sigaev)
- 为远程数据库访问添加 contrib/dblink (Joe Conway)
- contrib/ora2pg Oracle 转换工具 (Gilles Darold)
- contrib/xml XML 转换工具 (John Gray)
- contrib/fulltextindex 修复 (Christopher Kings-Lynne)
- 新增 contrib/fuzzystrmatch 带有 levenshtein 和 metaphone, soundex 合并 (Joe Conway)
- 添加contrib/intarray 布尔查询, 二分查找, 修复 (Oleg Bartunov)
- 新增 pg_upgrade 工具 (Bruce)
- 添加新的 pg_resetxlog 选项 (Bruce, Tom)

E.215. 版本 7.1.3

> 发布日期: 2001-08-15

E.215.1. 迁移到版本 7.1.3

运行7.1.X版本的系统不需要转储/恢复。

E.215.2. 修改列表

```
删除大事务未设置的 WAL 片段 (Tom)
Multiaction 规则修复 (Tom)
PL/pgSQL 内存分配修复 (Jan)
VACUUM 缓冲区修复 (Tom)
回归测试修复 (Tom)
在视图上为 GRANT/REVOKE/comments 修复了pg_dump, 用户定义类型 (Tom)
修复了带有 DISTINCT ON 或 LIMIT 的子查询 (Tom)
BeOS 修复
禁用 COPY TO/FROM 一个视图 (Tom)
Cygwin 建立 (Jason Tishler)
```

E.216. 版本 7.1.2

> 发布日期: 2001-05-11

这个版本对7.1.1有一个修补。

E.216.1. 迁移到版本 7.1.2

运行7.1.X的系统不需要转储/恢复。

E.216.2. 修改列表

```
当没有返回行时修复 PL/pgSQL SELECTs
修复 psql 反斜杠内核转储
参照完整性权限修复
优化器修复
pg_dump 清理
```

E.217. 版本 7.1.1

> 发布日期: 2001-05-05

这个版本对7.1有一些修补。

E.217.1. 迁移到版本 7.1.1

运行7.1的系统不需要转储/恢复。

E.217.2. 修改列表

```
修复数值型 MODULO 操作符 (Tom)
pg_dump 修复 (Philip)
pg_dump 可以转储 7.0 数据库 (Philip)
readline 4.2 修复 (Peter E)
JOIN 修复 (Tom)
AIX, MSWIN, VAX, N32K 修复 (Tom)
Multibytes 修复 (Tom)
Unicode 修复 (Tatsuo)
优化器改善 (Tom)
修复函数中的整行 (Tom)
修复 pg_ctl 和带有空格的选项字符串 (Peter E)
ODBC fixes (Hiroshi)
EXTRACT 现在可以接受字符串参数 (Thomas)
Python 修复 (Darcy)
```

E.218. 版本 7.1

> 发布日期: 2001-04-13

这个版本集中在移去多年来存在于PostgreSQL里的限制。

这个版本的主要修改：

预写式日志

在发生操作系统崩溃的情况下维持数据库系统稳定，以前版本的 PostgreSQL 在每次事务提交之前强迫所有数据修改都冲刷到磁盘上。有了 WAL， 只有一个日志文件必须冲刷到磁盘上，极大地提高了性能。 如果你在以前的版本曾经使用 -F 来关闭磁盘冲刷，那么你可以考虑不再使用这个选项。

TOAST

TOAST - 以前的版本有一个内编译的行长度限制，通常是 8k - 32k。 这个限制令我们很难存储长的文本域。有了 TOAST，任意长度的行都可以以很好的性能存储。

外连接

我们现在支持外连接。用于外连接的 UNION/NOT IN 的绕开现在不再需要了。 我们使用 SQL92 外连接语法。

函数管理器

以前的 C 函数管理器无法正确处理 NULL，也不支持 64 位 CPU (Alpha)。 新的函数管理器可以实现这些任务。你仍然可以继续使用你的老的自定义函数， 但是你在将来可能会愿意为使用新的函数管理器调用接口改写它们。

复杂查询

大量以前版本不支持的复杂查询现在可以运转了。许多视图，聚集，UNION，LIMIT， 游标， 子查询，和继承表的组合现在可以正确工作了。继承的表现在缺省是访问的。现在支持在 FROM 里的子查询了。

E.218.1. 迁移到版本 7.1

如果需要从任何以前版本升级到这个版本，那么要求用 pg_dump 做转储/恢复工作。

E.218.2. 修改列表

```
bug 修复
---------
许多multibyte/Unicode/locale修复 (Tatsuo and others)
更可靠的ALTER TABLE RENAME (Tom)
Kerberos V 修复 (David Wragg)
修复了有子查询的目标列表的INSERT INTO...SELECT (Tom)
标准错误提示用户名/密码 (Tom)
修复了to_char(), to_date(), to_ascii(), 和 to_timestamp() (Karel,
   Daniel Baldoni)
阻止查询表达式内存泄露 (Tom)
允许 UPDATE 数组元素 (Tom)
在取消时唤醒锁等待 (Hiroshi)
当使用哈希连接时修复极少情况的游标崩溃 (Tom)
在回滚的事务中修复 DROP TABLE/INDEX (Hiroshi)
如果启用了 MULTIBYTE, 修复 psql 的 \l+ 崩溃 (Peter E)
修复在 CREATE VIEW 运行时的规则名截断 (Ross Reedstrom)
修复PL/perl (Alex Kapranoff)
不允许 LOCK 视图 (Mark Hollomon)
不允许在视图上 INSERT/UPDATE/DELETE (Mark Hollomon)
不允许在视图上 DROP RULE, CREATE INDEX, TRUNCATE (Mark Hollomon)
允许 PL/pgSQL 接受非ASCII标识符 (Tatsuo)
允许视图适当处理 GROUP BY, aggregates, DISTINCT (Tom)
修复 TRUNCATE 命令极少的失败 (Tom)
允许 ALL, 子查询, 视图, DISTINCT, ORDER BY, SELECT...INTO 使用 UNION/INTERSECT/EXCEPT (Tom)
修复在中止事务期间的解析器错误 (Tom)
允许临时关系适当的清理索引 (Bruce)
在同一页中移动行以修复 VACUUM 问题 (Tom)
修改 pg_dump 以更好的处理在 template1 中用户定义的条目 (Philip)
允许在视图中使用 LIMIT (Tom)
要求游标准确的 FETCH LIMIT (Tom)
允许在非继承的字段上定义 PRIMARY/FOREIGN 键 (Stephan)
允许在子查询中 ORDER BY, LIMI (Tom)
允许在 CREATE RULE 中使用 UNION (Tom)
使 ALTER/DROP TABLE 可回滚 (Vadim, Tom)
在 pg_control 中存储 initdb 校对, 这样校对不能被改变 (Tom)
用规则修复 INSERT...SELECT (Tom)
修复视图内的 FOR UPDATE 和子查询 (Tom)
按照 SQL92 规则修复关于 NULL 的 OVERLAPS 操作符 (Tom)
修复 lpad() 和 rpad() 以处理长度小于输入字符串的情况 (Tom)
修复 NOTIFY 在一些规则中的使用 (Tom)
详细检查 btree 代码 (Tom)
用 Pl/pgSQL 变量修复 NOT NULL 的使用 (Tom)
详细检查 GIST 代码 (Oleg)
修复 CLUSTER 以保持约束和缺省字段 (Tom)
改进了死锁检测处理 (Tom)
在一个表中允许多个 SERIAL 字段 (Tom)
阻止偶然的索引损坏 (Vadim)

增强
------------
添加 OUTER JOINs (Tom)
彻底检查函数管理 (Tom)
允许在索引上 ALTER TABLE RENAME (Tom)
改善 CLUSTER (Tom)
为更多平台改善 ps 状态显示 (Peter E, Marc)
改善 CREATE FUNCTION 失败信息 (Ross)
JDBC 改进 (Peter, Travis Bauer, Christopher Cain, William Webber,
   Gunnar)
重要的统一配置模式/GUC。许多选项现在可以在 data/postgresql.conf,
postmaster/postgres 标识, 或 SET 命令中设置 (Peter E)
改善文件描述符缓存的处理 (Tom)
新增关于自动创建表别名的警告代码 (Bruce)
详细检查 initdb  进程 (Tom, Peter E)
详细检查继承的表;继承的表现在缺省是可访问的;新增 ONLY 关键字阻止对它的访问 (Chris Bitmead, Tom)
ODBC 清理/改善 (Nick Gorham, Stephan Szabo, Zoltan Kovacs,
   Michael Fork)
允许重命名临时表 (Tom)
详细检查内存管理环境 (Tom)
pg_dumpall 使用 CREATE USER 或 CREATE GROUP 而不是 COPY (Peter E)
详细检查 pg_dump (Philip Warner)
允许 pg_hba.conf 次要的口令文件只指定用户名 (Peter E)
```

```
在创建临时表时允许 TEMPORARY 或 TEMP 关键字 (Bruce)
新增内存泄露检查器 (Karel)
新增 SET SESSION CHARACTERISTICS (Thomas)
允许嵌套的块注释 (Thomas)
添加 WITHOUT TIME ZONE 类型限定符 (Thomas)
新增 ALTER TABLE ADD CONSTRAINT (Stephan)
为 INTEGER 聚集使用 NUMERIC 累加器 (Tom)
详细检查聚集代码 (Tom)
新增 VARIANCE 和 STDDEV() 聚集
改善 pg_dump 的依赖顺序 (Philip)
新增 pg_restore 命令 (Philip)
新增 pg_dump tar 输出选项 (Philip)
新增大对象的 pg_dump (Philip)
新增 ESCAPE 选项到 LIKE (Thomas)
新增大小写敏感的  LIKE - ILIKE (Thomas)
允许功能索引使用二进制兼容的类型 (Tom)
允许 SQL 函数在更多的环境中使用 (Tom)
新增 pg_config 工具 (Peter E)
新增 PL/pgSQL EXECUTE 命令，允许动态 SQL 和工具语句 (Jan)
为 SPI 值访问新增 PL/pgSQL GET DIAGNOSTICS 语句 (Jan)
新增 quote_identifiers() 和 quote_literal() 函数 (Jan)
新增 New ALTER TABLE table OWNER 到 user command (Mark Hollomon)
在 FROM 中允许子查询，如 FROM (SELECT ...) [AS] alias (Tom)
更新 PyGreSQL 到版本 3.1 (D'Arcy)
存储表为 OID 命名的文件 (Vadim)
新增 SQL 函数 setval(seq,val,bool) 适用于 pg_dump (Philip)
要求 DROP VIEW 删除视图，而不是 DROP TABLE (Mark)
允许 DROP VIEW view1, view2 (Mark)
在 DROP INDEX, DROP RULE, 和 DROP TYPE 中允许多个对象 (Tom)
允许自动转换到/从 Unicode (Tatsuo, Eiji)
新增 /contrib/pgcrypto 哈希函数 (Marko Kreen)
新增 pg_dumpall --globals-only 选项 (Peter E)
为创建新的 WAL 日志文件的 WAL 新增 CHECKPOINT 命令 (Vadim)
新增 AT TIME ZONE 语法 (Thomas)
允许配置 Unix 域套接字的位置 (David J. MacKenzie)
允许主进程监听一个特殊的 IP 地址 (David J. MacKenzie)
允许通过前导斜线以主机名指定套接字路径名 (David J. MacKenzie)
允许 CREATE DATABASE 指定临时数据库 (Tom)
新增工具以转换 MySQL 模式转储为 SQL92 和 PostgreSQL (Thomas)
新增 /contrib/rserv 复制工具包 (Vadim)
为  COPY BINARY 新增文件格式 (Tom)
新增 /contrib/oid2name 以映射数值文件到表名 (B Palmer)
新增 "idle in transaction" ps 状态信息 (Marc)
更新到 pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl 在关闭时现在缺省为 -w (wait)，  新增 -l (log) 选项
添加基本的依赖约束到 pg_dump (Philip)

类型
-----
修复 INET/CIDR 类型排序和添加新的函数 (Tom)
使 OID 表现为一个无符号的类型 (Tom)
允许 BIGINT  作为 INT8 的同义词 (Peter E)
新增 int2 和 int8 比较运算符 (Tom)
新增 BIT 和 BIT VARYING 类型 (Adriaan Joubert, Tom, Peter E)
CHAR()  因为 TOAST 而不再比 VARCHAR() 快 (Tom)
新增 GIST seg/cube 示例 (Gene Selkov)
改进 round(numeric) 处理 (Tom)
修复 CIDR 输出格式 (Tom)
新增 CIDR abbrev() 函数 (Tom)

性能
-----------
预写式日志 (WAL) 提供崩溃恢复，用较少的性能开销 (Vadim)
ANALYZE VACUUM 的阶段不再只锁住表 (Bruce)
减少文件搜索 (Denis Perchine)
为复制键改善 BTREE 代码 (Tom)
在一个单独的表中存储所有的大对象 (Denis Perchine, Tom)
改善内存分配性能 (Karel, Tom)

源码
-----------
新增函数管理调用约定 (Tom)
```

```
SGI 可移植性修复 (David Kaelbling)
新增 configure --enable-syslog 选项 (Peter E)
新增 BSDI README (Bruce)
配置脚本移动到顶级，而不是 /src (Peter E)
新增 configure --with-python 选项 (Peter E)
Solaris 清理 (Peter E)
详细检查 /contrib Makefiles (Karel)
新增 OpenSSL 配置选项 (Magnus, Peter E)
AIX 修复 (Andreas)
QNX 修复 (Maurizio)
新增 heap_open(), heap_openr() API (Tom)
删除 colon 和 semi-colon 运算符 (Thomas)
为视图新增 pg_class.relkind 值  (Mark Hollomon)
重命名 ichar() 为 chr() (Karel)
新增文件 btrim(), ascii(), chr(), repeat() (Karel)
修复 NT/Cygwin (Pete Forman)
AIX 端口 修复 (Andreas)
新增 BeOS 端口 (David Reid, Cyril Velter)
添加校对员的更改到文档 (Addison-Wesley, Bruce)
新增 Alpha 自旋锁代码 (Adriaan Joubert, Compaq)
详细检查 UnixWare 端口 (Peter E)
新增 Darwin/Mac OS X 端口 (Peter Bierman, Bruce Hartzler)
新增 FreeBSD Alpha 端口 (Alfred)
详细检查共享内存段 (Tom)
新增 IBM S/390 支持 (Neale Ferguson)
移动 macmanuf 到 /contrib (Larry Rosenman)
Syslog 改善 (Larry Rosenman)
新增不包含用户附加的 template0 数据库 (Tom)
新增 /contrib/cube 和 /contrib/seg GIST 示例代码 (Gene Selkov)
允许 NetBSD 的 libedit 而不是 readline (Peter)
改善汇编语言源代码格式 (Bruce)
新增 contrib/pg_logger
新增 --template 选项到 createdb
新增 contrib/pg_control 工具 (Oliver)
新增 FreeBSD 工具 ipc_check, start-scripts/freebsd
```

E.219. 版本 7.0.3

> 发布日期: 2000-11-11

有一些对7.0.2的修补。

E.219.1. 迁移到版本 7.0.3

运行7.0.*的系统不需要转储/恢复。

E.219.2. 修改列表

```
Jdbc 修复 (Peter)
大对象修复 (Tom)
修复 lean 在 COPY WITH OIDS 时的泄露 (Tom)
修复 backwards-index-scan (Tom)
修复 SELECT ... FOR UPDATE, 这样它检查重复的键 (Hiroshi)
添加 --enable-syslog 到配置 (Marc)
修复极少情况下后端退出时的中止事务 (Tom)
启用多字节时修复 psql \l+ (Tatsuo)
允许 PL/pgSQL 接受非ascii标识符 (Tatsuo)
使 vacuum 总是刷新缓冲区 (Tom)
当等待一个锁时允许取消 (Hiroshi)
修复用户认证代码中的内存分配问题 (Tom)
删除伪造的 int4out() 的使用 (Tom)
修复 COALESCE 或 BETWEEN 中的多重子查询 (Tom)
修复在特定情况下打开的堆中的触发器错误 (Jeroen van Vianen)
修复错误的不相等的选择性 (Tom)
修复错误的 strcmp() 的使用 (Tom)
修复存储管理访问条目超过文件结尾的bug (Tom)
修复在所有 smgr elog 消息中包括内核错误信息 (Tom)
修复在建立时 '.' 不在 PATH 中 (SL Baur)
修复超出文件描述符错误 (Tom)
修复使 pg_dump 为函数转储 'iscachable' 标记 (Tom)
修复在附加节点的目标列表中的子查询 (Tom)
修复合并连接规划 (Tom)
修复带有索引的关系的 TRUNCATE 失败 (Tom)
避免写入误差时数据库范围的重启 (Hiroshi)
修复通过重新计算输出使 nodeMaterial 遵守 chgParam (Tom)
修复当行版本的出发点和目的地位于同一页面时, 更新行版本的移动链条的 VACUUM 问题 (Tom)
修复 user.c CommandCounterIncrement (Tom)
修复 AM/PM 在 to_char() 中的边界问题 (Karel Zak)
修复 TIME 聚集处理 (Tom)
修复 to_char() 避免 NULL 输入的内核转储 (Tom)
缓存修复 (Tom)
修复插入/拷贝更长的多字节字符串到 char() 数据类型 (Tatsuo)
修复中止时的后端崩溃 (Tom)
```

E.220. 版本 7.0.2

> 发布日期: 2000-06-05

这是对7.0.1和添加的文档的重新打包。

E.220.1. 迁移到版本 7.0.2

运行7.*的系统不需要转储/恢复。

E.220.2. 修改列表

```
添加文档到原始代码。
```

E.221. 版本 7.0.1

> 发布日期: 2000-06-01

这是一个7.0的纯净版本。

E.221.1. 迁移到版本 7.0.1

运行7.0版本的系统不需要转储/恢复。

E.221.2. 修改列表

```
修复许多 CLUSTER 失败 (Tom)
允许 ALTER TABLE RENAME 在索引上工作 (Tom)
修复 plpgsql 处理 datetime->timestamp 和 timespan->interval (Bruce)
新增 configure --with-setproctitle 开关以使用 setproctitle() (Marc, Bruce)
修复因为一个来自6.5.3或更多的结果集的错误而关闭的问题
jdbc 结果集修复 (Joseph Shraibman)
优化器优化 (Tom)
修复为 pgaccess 创建用户
修复 UNLISTEN 失败
IRIX 修复 (David Kaelbling)
QNX 修复 (Andreas Kardos)
降低 COPY IN 锁级别 (Tom)
改变 libpqeasy 以使用 PQconnectdb() 风格参数 (Bruce)
修复 pg_dump 以处理 OID 索引 (Tom)
修复小的内存泄露 (Tom)
修复 Solaris 以 createdb/dropdb (Tatsuo)
修复非阻塞连接 (Alfred Perlstein)
修复 RENAME TABLE 失败之后不正确的恢复 (Tom)
在安装中拷贝 pg_ident.conf.sample 到 /lib 目录 (Bruce)
添加 SJIS UDC (NEC selection IBM kanji) 支持 (Eiji Tokuya)
修复太长的系统信息 (Tatsuo)
修复引用的索引太长的问题 (Tom)
JDBC ResultSet.getTimestamp() 修复 (Gregory Krasnow & Floyd Marinescu)
ecpg 改变 (Michael)
```

E.222. 版本 7.0

> 发布日期: 2000-05-08

这个版本包含许多方面的改进，表现了 PostgreSQL 的不断成长。在7.0中有比任意以前版本更多的改进和修复。开发人员相信这是目前最好的版本； 我们尽力做到只发布稳固的版本，这个版本也不例外。

这个版本中的主要变化：

外键

现在我们已经实现了Foreign keys（外键）， 只有 PARTIAL MATCH 外键没有实现。 许多用户要求增加这个特性，而我们很高兴现在我们能提供这个特性.

优化器检修

继续一年前开始的工作，优化器现在已经进步多了，允许更好的查询规划选择和更好的性能，而且内存用得更少。

升级了psql

psql，是我们的交互终端监控器，升级并增加了许多新特性。 参阅 psql 手册页获取细节。

连接语法

我们现在支持 SQL92 连接（JOIN）语法了，尽管目前的版本只支持 `INNER JOIN` 。 `JOIN` ， `NATURAL JOIN` ， `JOIN` / `USING` ， `JOIN` / `ON` 都可用，这些是字段集合名称.

E.222.1. 迁移到版本 7.0

那些从任何以前版本的 PostgreSQL 移植的人都需要用 pg_dump 做一次转储/恢复工作。对于从 6.5.* 升级的用户， 你可以使用 pg_upgrade 升级到这个版本；不过， 升级最稳固的方法还是一次完整的转储/重载安装。

这个新版本要考虑的接口和兼容性问题包括：

- 日期/时间类型 `datetime` 和 `timespan` 已经被 SQL92 定义类型 `timestamp` 和 `interval` 取代了。管我们做了一些努力令 PostgreSQL 可以识别这些过时的类型名并把它们转换成新类型名以简化类型转化， 但是这个机制可能不完全对你的现有应用透明。
- 优化器在查询开销计算方面已经有了显著的提升。有时，这样将令查询时间减少， 因为优化器做出了查询规划的更好选择。不过，有一小部分情况下， 通常是数据不正常（病态）分布时，你的查询时间可能增加。如果你处理大量的数据， 你可能要检查你的查询

以核实性能.

- JDBC 和 ODBC 接口都升级和扩展了。
- 字符串函数 `CHAR_LENGTH` 现在是一个内部函数。 以前版本是把它转换成对 `LENGTH` 的调用， 这样做会导致与其他类型实现的 `LENGTH` 的混淆，比如几何类型.

E.222.2. 修改列表

```
 Bug 修复
---------
阻止函数调用超出参数的最大数目 (Tom)
改善 CASE 构造 (Tom)
修复 SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
修复 SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
修复 GROUP BY 扫描 bug (Tom)
改进 SQL 语法处理 (Tom)
修复包含在 INSERT ... SELECT ... 里的视图 (Tom)
修复 SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
修复 subselects in INSERT ... SELECT (Tom)
阻止 INSERT ... SELECT ... ORDER BY (Tom)
修复大于 2GB 的关系，包括 vacuum
提高传播系统表更改为其他后端 (Tom)
提高传播用户表更改为其他后端 (Tom)
修复在复杂的环境中处理临时表 (Bruce, Tom)
在表打开时允许表锁，提高并发可靠性 (Tom)
在 pg_dump 中适当的引用序列名 (Ross J. Reedstrom)
在有其他访问时阻止 DROP DATABASE
如果没有行被处理那么阻止 GROUP BY 返回任何行 (Tom)
如果没有行匹配 WHERE 那么修复 SELECT COUNT(1) FROM table WHERE ...' (Tom)
修复 pg_upgrade 使其为 MVCC 工作 (Tom)
修复 SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
修复 "f1 datetime DEFAULT 'now'"  (Tom)
修复 DEFAULT 中使用的 CURRENT_DATE 的问题 (Tom)
允许只有注释的行，也允许 ;;; 行 (Tom)
改进在磁盘满写入磁盘失败之后的恢复 (Hiroshi)
修复在 FROM 中提到表而没有连接该表的情况 (Tom)
允许 HAVING 子句不带有聚集函数 (Tom)
修复 "--" 注释和没有尾随的新行，就像在 Perl 接口中看到的那样
改善 pg_dump 失败错误报告 (Bruce)
允许排序和哈希超过 2GB 文件大小 (Tom)
修复 pg_dump 转储非继承的规则 (Tom)
修复 NULL 处理比较 (Tom)
修复由于 CREATE/DROP 命令失败引起的不一致的状态 (Hiroshi)
修复带有破折号的 dbname
阻止 DROP INDEX 干扰其他后端 (Tom)
修复 verify_password() 中的文件描述符泄露
修复 "Unable to identify an operator =$" 问题
修复 ODBC，这样如果启用了 CommLog 和 Debug 不会有 segfault  (Dirk Niggemann)
修复递归的退出调用 (Massimo)
修复超长的时区 (Jeroen van Vianen)
使 pg_dump 保存主键信息 (Peter E)
阻止带有单引号的数据库 (Peter E)
阻止在事务内部 DROP DATABASE (Peter E)
ecpg 内存泄露修复 (Stephen Birch)
修复 SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)
Y2K 时间戳修复 (Massimo)
修复 VACUUM 'HEAP_MOVED_IN 不是预期的的错误 (Tom)
修复带有包含空格的表/字段的视图 (Tom)
阻止在索引上的权限 (Peter E)
修复产生错误时自旋锁卡住的问题 (Hiroshi)
修复 Linux 上的 ipcclean
修复 NULL 约束条件的处理 (Tom)
修复 odbc 驱动中的内存泄露 (Nick Gorham)
修复 UNION 表上的权限检查 (Tom)
```

```
修复以允许 SELECT 'a' LIKE 'a' (Tom)
修复 SELECT 1 + NULL (Tom)
修复 CHAR
修复数值类型上的 log() (Tom)
反对 ':' 和 ';' 操作符
允许 vacuum 临时表
不允许继承的字段和新字段的名字相同
当磁盘空间被耗尽时恢复或强制失败 (Hiroshi)
修复 AS 字段匹配结果字段的 INSERT INTO ... SELECT
修复 INSERT ... SELECT ... GROUP BY 以目标字段分组而不是以源字段分组 (Tom)
修复 CREATE TABLE test (a char(5) DEFAULT text '', b int4) with INSERT (Tom)
修复带有 LIMIT 的 UNION
修复 CREATE TABLE x AS SELECT 1 UNION SELECT 2
修复 CREATE TABLE test(col char(2) DEFAULT user)
修复 CREATE TABLE ... DEFAULT 中不匹配的类型
修复 SELECT * FROM pg_class where oid in (0,-1)
修复 SELECT COUNT('asdf') FROM pg_class WHERE oid=12

增强
------------
新增 CLI 接口包括文件 sqlcli.h, 基于 SQL3/SQL98
删除所有在查询长度上的限制，行的长度限制仍然存在 (Tom)
更新 jdbc 协议到 2.0 (Jens Glaser <[jens@jens.de](mailto:jens@jens.de)>)
添加 TRUNCATE 到快速截断关系 (Mike Mascari)
修复以给予超级用户和 createdb 用户适当更新目录的权限 (Peter E)
允许 ecpg 布尔变量有 NULL 值 (Christof)
如果变量的 NULL 值带有非 NULL 指示符那么发出 ecpg 错误 (Christof)
允许 ^C 取消 COPY 命令 (Massimo)
添加 SET FSYNC 和 SHOW PG_OPTIONS 命令(Massimo)
为动态加载的 C 函数重载函数名 (Frankpitt)
添加 CmdTuples() 到 libpq++(Vince)
新增 CREATE CONSTRAINT TRIGGER 和 SET CONSTRAINTS 命令(Jan)
允许 CREATE FUNCTION/WITH 子句用于所有的语言类型
configure --enable-debug 添加 -g (Peter E)
configure --disable-debug 删除 -g (Peter E)
允许更复杂的缺省表达式 (Tom)
第一个真正的 FOREIGN KEY 约束触发器功能 (Jan)
添加 FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
添加 FOREIGN KEY ... MATCH <unspecified> 参照操作 (Don Baccus)
允许在 ctid(堆的物理位置) 上的 WHERE 约束 (Hiroshi)
从贡献包中移动 pginterface 到接口目录，重命名为 pgeasy (Bruce)
改变 pgeasy connectdb() 参数顺序 (Bruce)
要求 SELECT DISTINCT 目标列表拥有所有 ORDER BY 字段 (Tom)
添加 Oracle 的 COMMENT ON 命令 (Mike Mascari <[mascarim@yahoo.com](mailto:mascarim@yahoo.cc
libpq 的 PQsetNoticeProcessor 函数现在返回先前的 hook(Peter E)
阻止 PQsetNoticeProcessor 被设置为 NULL (Peter E)
在 COPY 选项中使 USING 可选 (Bruce)
允许在目标列表中有子查询 (Tom)
允许子查询在比较操作符的左边 (Tom)
新增并行回归测试 (Jan)
改变后端的 COPY 写入文件权限为 644 而不是 666 (Tom)
强制 PGDATA 目录上的权限为安全的，即使它仍然存在 (Tom)
添加 psql LASTOID 变量以返回最后继承的 oid (Peter E)
允许并发的 vacuum 和删除 pg_vlock vacuum 锁文件 (Tom)
为 vacuum 添加权限检查 (Peter E)
新增 libpq 函数以允许异步的连接：PQconnectStart(),
  PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
  PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
新增 libpq PQsetenv() 函数 (Ewan Mellor)
创建/更改 用户扩展 (Peter E)
在 $PGDATA 下新增 postmaster.pid 和 postmaster.opts (Tatsuo)
为创建/删除 用户/数据库 新增脚本 (Peter E)
详细检查主要的 psql  (Peter E)
添加常量到 libpq 接口 (Peter E)
新增 libpq 函数 PQoidValue (Peter E)
显示特定的非聚集导致的 GROUP BY 问题 (Tom)
改变 pg_shadow 重新创建 pg_pwd 文件 (Peter E)
添加 aggregate(DISTINCT ...) (Tom)
允许标识控制  NULL 的 COPY 输入/输出 (Peter E)
使 postgres 用户有缺省的口令 (Peter E)
添加 CREATE/ALTER/DROP GROUP (Peter E)
所有管理脚本现在支持 --long 选项 (Peter E, Karel)
```

```
Vacuumdb 脚本现在支持 --all 选项 (Peter E)
ecpg 新增轻便的 FETCH 语法
添加 ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
        和 EXEC SQL ENDIF 指令
添加 pg_ctl 脚本以控制后端启动 (Tatsuo)
添加 postmaster.opts.default 文件以存储启动标识 (Tatsuo)
允许 --with-mb=SQL_ASCII
增加索引键的最大数量到 16 (Bruce)
增加函数参数的最大数量到 16 (Bruce)
允许配置索引键和参数的最大数量 (Bruce)
允许非特权的用户改变他们的口令 (Peter E)
启用口令认证；新用户需要 (Peter E)
不允许删除用于数据库的用户 (Peter E)
改变 initdb 选项 --with-mb 为 --enable-multibyte
添加 initdb 选项以为超级用户提示口令 (Peter E)
允许像 col::numeric(9,2) 和 col::int2::float8 这样的复杂类型转换 (Tom)
在 initdb, initlocation, pg_dump, ipcclean 上升级用户接口 (Peter E)
新增 pg_char_to_encoding() 和 pg_encoding_to_char() 函数 (Tatsuo)
libpq 非阻塞模式 (Alfred Perlstein)
改善在没有声明长度的计算中的类型的转换
新增 plperl 内部编程语言 (Mark Hollomon)
允许 COPY IN 读取不以一个新行结束的文件 (Tom)
当长的标识符被截断时提示 (Tom)
允许聚集使用类型等价 (Peter E)
允许 Oracle 的 to_char(), to_date(), to_datetime(), to_timestamp(), to_number() 转换函数 (Ka
添加 SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
检查以确保 ORDER BY 和 DISTINCT 操作一致 (Tom)
添加 NUMERIC 和 int8 类型到 ODBC
改善 Append, Group, Agg, Unique 的 EXPLAIN 结果 (Tom)
添加 ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
在 PL/pgSQL 中允许 SELECT .. FOR UPDATE (Hiroshi)
启用后端顺序扫描，即使到达了 EOF (Hiroshi)
添加布尔值的 btree 索引，>= 和 <= (Don Baccus)
当 COPY FROM 失败时打印当前行的编号 (Massimo)
识别出 POSIX 时区，如 "PST+8" 和 "GMT-8" (Thomas)
添加 DEC 作为 DECIMAL 的同义词 (Thomas)
添加 SESSION_USER 作为 SQL92 的关键字，和 CURRENT_USER 相同 (Thomas)
改善 SQL92 字段别名 (aka 相关名字) (Thomas)
改善 SQL92 连接语法 (Thomas)
使 INTERVAL 的保留字作为一个字段标识符被允许 (Thomas)
实现 REINDEX 命令 (Hiroshi)
在聚集函数 SUM(ALL col) 中接受 ALL (Tom)
阻止 GROUP BY 使用字段别名 (Tom)
新增 psql \encoding 选项 (Tatsuo)
当在 waiting-for-lock 状态时允许 PQrequestCancel() 终止 (Hiroshi)
在所有情况下允许负数的否定
添加 ecpg 描述符 (Christof, Michael)
允许 CREATE VIEW v AS SELECT f1::char(8) FROM tbl
允许带有长度的转换，如 foo::char(8)
新增 libpq 函数 PQsetClientEncoding(), PQclientEncoding() (Tatsuo)
添加 SJIS 用户定义字符支持 (Tatsuo)
大的视图/规则支持
使 libpq 的 PQconndefaults() 线程安全 (Tom)
禁用 // 作为注释以符合 ANSI，应该使用 -- (Tom)
允许在视图的 CREATE VIEW name 上使用字段别名 (collist)
修复带有子查询的视图 (Tom)
允许 UPDATE table SET fld = (SELECT ...) (Tom)
SET 命令选项不再需要引号
升级 pgaccess 到 0.98.6
新增SET SEED 命令
新增 pg_options.sample 文件
新增 SET FSYNC 命令 (Massimo)
当创建表时允许 pg_descriptions
当创建类型、字段和函数时允许 pg_descriptions
允许 psql \copy 以允许分隔符 (Peter E)
允许 psql 打印 null 以区别 "" [null] (Peter E)

类型
-----
修复了许多数组 (Tom)
允许空的字段名作为数组的下标 (Tom)
改善 int 和 float 常量的类型转换  (Tom)
```

```
清理 int8 输入、范围检查和类型转换 (Tom)
修复 SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') 是 255.255.255.255 而不是 0.0.0.0 (Oleg Sharoiko)
在 NUMERIC 上添加 btree 索引 (Jan)
含有 NUL 字符的大对象的 perl 修复 (Douglas Thomson)
大对象的 ODBC 修复 (free)
修复 cidr 数据类型的索引
修复以太网 MAC 地址 (macaddr 类型)比较
当在计算中发生溢出时修复日期/时间类型 (Tom)
允许 int8 上的数组 (Peter E)
修复 NUMERIC 类型的舍入/溢出，如 NUMERIC(4,4) (Tom)
允许 NUMERIC 数组
修复 NUMERIC ceil() 和 floor() 函数中的 bug (Tom)
使 char_length()/octet_length 包含结尾空白 （Tom）
使 abstime/reltime 使用 int4 而不是 time_t (Peter E)
为压缩的文本字段新增 lztext 数据类型
修正代码以处理强制的 int 和 float 常量 (Tom)
在新的代码开始实现 BIT 和 BIT VARYING 类型 (Adriaan Joubert)
NUMERIC 现在接受科学计数法 (Tom)
NUMERIC 到 int4 的圆整 (Tom)
适当的转换 float4/8 到 NUMERIC (Tom)
允许 NUMERIC 的类型转换 (Thomas)
使 ISO 数据类型 (2000-02-16 09:33) 为缺省 (Thomas)
添加 NATIONAL CHAR [ VARYING ] (Thomas)
允许 NUMERIC 圆整和截断以接受负的标度 (Tom)
新增 TIME WITH TIME ZONE 类型 (Thomas)
在时间类型上添加 MAX()/MIN() (Thomas)
为 int8 添加 abs(), mod(), fac() (Thomas)
为 float8 重命名函数为 round(), sqrt(), cbrt(), pow() (Thomas)
为 float8 添加超越数学函数(如 sin(), acos()) (Thomas)
为 NUMERIC 类型添加 exp() 和 ln()
重命名 NUMERIC power() 为 pow() (Thomas)
改进了 TRANSLATE() 函数 (Edwin Ramirez, Tom)
允许 X=-Y 运算符  (Tom)
允许 SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
允许 LOCALE 在正则表达式搜索中使用索引 (Tom)
允许功能性索引的创建使用缺省的类型

性能
-----------
阻止带有许多 AND 和 OR 的指数空间消耗 (Tom)
收集系统字段的属性选择值 (Tom)
减少聚集的内存使用 (Tom)
修复 LIKE 最优化以使用带有多字节编码的索引 (Tom)
修复 r-tree 索引优化器选择性 (Thomas)
改善优化器选择性计算和功能 (Tom)
优化许多相等键存在的 btree 搜索 (Tom)
只在索引存在时启用快速 LIKE 索引进程 (Tom)
再次利用带有多重记录的索引页的自由空间 (Tom)
改善哈希连接进程 (Tom)
如果结果已经排序，那么阻止降序排序 (Hiroshi)
允许索引扫描查询限制条件的交换 (Tom)
在需要 ORDER BY/GROUP BY 的情况下更喜欢索引扫描 (Tom)
为性能分配大量内存请求固定尺寸的块 (Tom)
通过减少内存分配请求修复 vacuum 的性能 (Tom)
实现常量表达式简化 (Bernard Frankpitt, Tom)
使用使用的第二字段来决定索引扫描的开始  (Hiroshi)
在做内部排序时阻止四倍的使用磁盘空间 (Tom)
通过调用更少的函数快速的排序 (Tom)
创建系统索引匹配所有系统缓存 (Bruce, Hiroshi)
使系统缓存使用系统索引 (Bruce)
使所有系统索引唯一 (Bruce)
为 vacuum 速度提升改进 pg_statistics 管理 (Tom)
较低频率的刷新后端缓存 (Tom, Hiroshi)
COPY 现在重新使用了先前的内存分配，提高了性能 (Tom)
改善优化成本估算 (Tom)
改善优化器估算  x > lowbound AND x < highbound  的范围查询 (Tom)
在适当的地方使用 DNF 而不是 CNF (Tom, Taral)
进一步清理 OR-of-AND WHERE-clauses (Tom)
在 OR 子句 (x = 1 AND y = 2) OR (x = 2 AND y = 4) 中使用索引 (Tom)
智能优化器计算随机索引页访问 (Tom)
新增 SET 变量控制优化器开销 (Tom)
```

```
优化器查询基于 LIMIT, OFFSET, 和 EXISTS 限制条件 (Tom)
减少优化器链接路径的内部开支以加速 (Tom)
主要的子查询加速 (Tom)
当没有禁用 fsync 时较少的 fsync 写入 (Tom)
改善 LIKE 优化器估算  (Tom)
在只有 SELECT 的查询中阻止 fsync (Tom)
使索引创建使用 psort 代码, 因为现在它更快速 (Tom)
允许创建临时表 > 1 Gig 的排序

源代码树的变化
-------------------
修复 linux PPC 编译
新增一般的 expression-tree-walker 子程序 (Tom)
更改 form() 为 varargform() 以阻止可能的问题
为 Alphas 上的大整数改善范围检查
清除 /include 目录中的 #include (Bruce)
为检查包含添加脚本 (Bruce)
从 *.c 文件中删除不需要的 #include (Bruce)
更改 #include 按情况使用 <> 和 "" (Bruce)
启用 Windows 的 libpq 编译
来自 Uncle George <[gatgul@voicenet.com](mailto:gatgul@voicenet.com)> 的 Alpha 自旋锁修复
彻底检修优化器数据结构 (Tom)
修复 cygipc 库 (Yutaka Tanida)
允许 pgsql 工作在新的 Cygwin 快照上 (Dan)
新增目录版本号 (Tom)
添加 Linux ARM
重命名 heap_replace 为 heap_update
更新 QNX (Dr. Andreas Kardos)
新增特定于平台的回归处理 (Tom)
重命名 oid8 -> oidvector 和 int28 -> int2vector (Bruce)
包含所有 yacc 和 lex 文件到发布中 (Peter E.)
删除不再需要的 lextest (Peter E)
修复在 Windows 上的 libpq 和 psql (Magnus)
内部改变 datetime 和 timespan 为 timestamp 和 interval (Thomas)
在 BSD/OS 上修复 plpgsql
添加 SQL_ASCII 测试案例到回归测试 (Tatsuo)
configure --with-mb 现在废弃了 (Tatsuo)
修复了 NT
修复了 NetBSD (Johnny C. Lam <[lamj@stat.cmu.edu](mailto:lamj@stat.cmu.edu)>)
修复 Alpha 编译
新增多字节编码
```

E.223. 版本 6.5.3

> 发布日期: 1999-10-13

这是基本上是一个 6.5.2 的净化版。我们添加了一些在 6.5.2 中丢失的 PgAccess， 并且安装了一个 NT 特定的补丁。

E.223.1. 迁移到版本 6.5.3

运行 6.5.* 的用户不需要转储/恢复。

E.223.2. 修改列表

```
升级版本的 pgaccess 0.98
NT 相关的补丁
修补了在继承表里的转储规则
```

E.224. 版本 6.5.2

> 发布日期: 1999-09-15

这基本是一个6.5.1的净化版。我们修复了许多 6.5.1 的用户报告的问题。

E.224.1. 迁移到版本 6.5.2

运行 6.5.* 的用户不需要转储/恢复。

E.224.2. 修改列表

```
修复了 subselect+CASE (Tom)
为 solaris_i386 和 solaris_sparc 端口添加了 SHLIB_LINK 设置 (Daren Sefcik)
修复了在 WHERE 连接子句中的 CASE (Tom)
修复了 BTScan 中止 (Tom)
修复对多余的 UNIQUE 和 PRIMARY KEY 索引的检查 (Thomas)
改善使其检查多字段约束 (Thomas)
修复 Windows 下启用 MB 的问题 (Hiroki Kataoka)
允许 BSC yacc 和 bison 编译 pl 代码 (Bruce)
修复 SET NAMES 工作
修复了 int8 (Thomas)
修复 vacuum 的内存消耗 (Hiroshi,Tatsuo)
减少 vacuum 的总内存消耗 (Tom)
修复 timestamp(datetime)
规则逆句法分析 bug 修复 (Tom)
修复 mkMakefile.tcldefs.sh.in 和 mkMakefile.tkdefs.sh.in 中的引用问题 (Tom)
重新使用 vacuum 在索引页上释放的空间 (Vadim)
为 pg_dump 记录 -x (Bruce)
修复规则逆分析器中的一元操作符 (Tom)
注释掉在 mdtruncate() 期间超过分段的 FileUnlink (Tom)
修复 LIKE 中的逻辑错误：当在到达文本的结尾之前到达模式的结尾时不应该返回 LIKE_ABORT (Tom)
修复在事务中止期间对堆栈内存分配的不正确的清理 (Tom)
升级 pgaccess 0.98 的版本
```

E.225. 版本 6.5.1

> 发布日期: 1999-07-15

这基本上是一个6.5的净化版本。我们修复了一系列6.5用户报告的问题。

E.225.1. 迁移到版本 6.5.1

运行6.5的用户不需要转储/恢复。

E.225.2. 修改列表

```
添加 NT README 文件
linux_ppc, IRIX, linux_alpha, OpenBSD, alpha 的可移植性修复
删除 QUERY_LIMIT, 使用 SELECT...LIMIT
修复 EXPLAIN 的继承 (Tom)
允许在多节的表上 vacuum 的补丁 (Hiroshi)
R-Tree 优化器选择性修复 (Tom)
ACL 文件描述符泄露修复 (Atsushi Ogawa)
新增表达式子树代码 (Tom)
避免只读事务的磁盘写入 (Vadim)
如果最后一个事务失败了修复对临时表的删除 (Bruce)
修复了阻止创建太大的行 (Bruce)
plpgsql 修复
允许端口号的范围为 32k - 64k(Bruce)
添加 ^ 优先级 (Bruce)
重命名分类文件 pg_temp 为 pg_sorttemp (Bruce)
修复时间值中的微秒 (Tom)
教程来源清理
新增 linux_m68k 端口
修复 NULL 在某些情况下的排序 (Tom)
共享库依赖性修复 (Tom)
固定故障影响子查询中的 GROUP BY (Tom)
修复了一些编译器警告 (Tomoaki Nishiyama)
添加 Win1250 (Czech) 支持 (Pavel Behal)
```

E.226. 版本 6.5

> 发布日期: 1999-06-09

这个版本标志着开发队伍对从伯克利继承过来的代码的掌握和理解达到了一个新的阶段。 你将看到现在我们更容易增加新的特性，这些得益于我们全世界开发队伍的不断壮大和经验的不断丰富。

下面是一些最引人注意的改变的简介：

多版本并行控制（MVCC）

这个东西废止了我们老式的表级别的锁，取而代之的是一个比大多数商用数据库系统都先进的锁系统。 在传统的系统里，每行的修改是先锁住，直到提交，以此避免被其他用户读取。 MVCC 利用 PostgreSQL 天生的多版本特性允许读者在写活动中继续读取一致的数据。 写入者继续使用紧凑的 pg_log 事务系统。所有这些都是在不需要为每行分配一个锁 （像传统的数据库系统那样）的情况下进行的。 因此，基本上我们不再受制于简单的表级别锁；我们有比行级别锁更好的技术。

pg_dump 的热备份

pg_dump 利用了新的 MVCC 特性， 可以在数据库保持在线和可以执行查询的情况下，进行一致的数据转储/备份。

数值数据类型

我们现在有了真正的数值数据类型，可以由用户定义精度。

临时表

我们保证临时表在一次数据库会话过程中唯一，并且在会话结束时删除。

新的 SQL 特性

我们现在有了 CASE，INTERSECT 和 EXCEPT 语句支持。我们有了新的 LIMIT/OFFSET，SET TRANSACTION ISOLATION LEVEL，SELECT ... FOR UPDATE 和一个改进了的 LOCK TABLE 命令。

提速

感谢我们队伍里的许多聪明的头脑，我们继续给 PostgreSQL 提速。 我们加速了存储器分配，优化，表联合以及行传输过程。

端口

我们继续扩展我们的端口列表，这回包括了 Windows NT/ix86 and NetBSD/arm32。

界面

大多数界面有了新的版本，现有的功能也被改进了。

文档

在这个文档里所有地方都有新的和更新了的内容。 新的 FAQ 已经分配到了 SGI 和 AIX 平台。 教程里包含了来自 Stefan Simkovics 的关于 SQL 的介绍性信息。对于用户手册，我们包括了关于 postmaster 和更多工具程序的参考内容， 还有一个描述日期时间详细特性的附件。管理员手册 包含了一个 Tom Lane 的新的关于错误分析的章节。程序员手册 包括了 Stefan 写的查询处理的描述，以及通过匿名 CVS 和 CVSup获取 PostgreSQL 源码树的详细内容。

E.226.1. 迁移到版本 6.5

要想从以前的 PostgreSQL 版本迁移到新的版本， 我们需要用 pg*dump* 进行转储/恢复的工作。 不能_用 pg_upgrade 升级到这个版本， 因为和以前版本相比，表在磁盘上的（存储）结构已经经过修改了。

新的多版本并发控制（MVCC）特性在多用户环境里可能有一些不同的表现。 阅读并理解下面段落，确保你现有的应用将表现出你所希望的特性。

E.226.1.1. 多版本并发控制

因为不管事务的隔离级别是什么，6.5 里的读动作不锁定数据， 一个事务读的数据可能被其他事务覆盖。换句话说，如果 `SELECT` 返回了一行并不意味着该行在被返回的时候（也就是说在语句或者事务开始后的某个时间）真正存在， 也不意味着在当前事务做提交或者回滚之前该行被保护免于被并行事务删除或者修改。

要保证一行的确实存在并且保护它免于被并行更新，我们必须使用 `SELECT FOR UPDATE` 或一个合适的 `LOCK TABLE` 语句。我们在从以前的 PostgreSQL 和其他环境移植应用时必须考虑这一点。

如果你正用 `contrib/refint.*` 触发器保证参考完整性， 你就必须记住上面几条。现在还需要额外的技巧。一个方法是： 如果一个事务准备更新/删除一个主键时，使用 `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` 命令， 如果一个事务准备更新/插入一个外键时使用 `LOCK parent_table IN SHARE MODE` 命令。

> **Note:** 要注意如果你运行一个处于 SERIALIZABLE 模式下的事务，在该事务里， 你必须在执行任何 DML 语句 （`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`） 之前执行上面提到的 `LOCK` 命令。

这些不便将在今后我们实现了读脏（未提交）数据的能力（不管什么隔离级别）和真正的参考完整性后消失。

E.226.2. 修改列表

```
Bug 修复
---------
修复 text<->float8 和 text<->float4 转换函数(Thomas)
修复带有混合约束的情况创建表 (Billy)
改变 exp()/pow() 的行为以在下溢/溢出时产生错误 (Jan)
修复 pg_dump -z 中的 bug
内存溢出清理 (Tatsuo)
修复 lo_import 崩溃 (Tatsuo)
调整数据类型名的处理以抑制双引号 (Thomas)
使用类型转换匹配字段和 DEFAULT (Thomas)
修复死锁，这样它只在一秒钟的休眠之后检查一次 (Bruce)
修复聚集和 PL/pgsql (Hiroshi)
修复子查询崩溃 (Vadim)
修复 libpq 函数 PQfnumber 和大小写敏感的名字 (Bahman Rafatjoo)
修复大对象的 write-in-middle，没有额外的阻塞、内存消耗 (Tatsuo)
修复 pg_dump -d 或 -D 和在 INSERT 中引用特殊字符
修复 dynahash 的严重的问题 (Tom)
修复 INET/CIDR 可移植性问题
修复 ALTER TABLE ADD COLUMN 中的可选择性错误问题 (Bruce)
修复执行器使不同字段类型的合并连接能够工作 (Tom)
修复 Alpha OR 选择性 bug
修复 OR 索引选择性问题 (Bruce)
修复 so \d 显示 char()/varchar() 的固有长度 (Ryan)
修复教程代码 (Clark)
提高 destroyuser 检查 (Oliver)
修复 Kerberos (Rodney McDuff)
修复当缓冲区脏了时删除数据库 (Bruce)
修复 nextval() 序列可以大小写敏感 (Bruce)
修复 !!= 运算符
在破坏数据库文件之前删除缓冲区 (Bruce)
修复执行器估算函数两次的情况 (Tatsuo)
允许序列的下一行动作大小写敏感 (Bruce)
修复优化器索引不为负数工作 (Bruce)
修复 fjIsNull 执行器内的内存泄露
修复聚集的内存泄露 (Erik Riedel)
允许用户名包含一个破折号来授予权限
清理 inet 类型中的 NULL
清理系统表 bug (Tom)
修复 PAGER 和 \? 命令的问题 (Masaaki Sakaida)
降低缺省多段文件大小限制为 1GB (Peter)
修复 CREATE OPERATOR 的转储 (Tom)
修复游标的向后扫描 (Hiroshi Inoue)
修复使用 \i 时的 COPY FROM STDIN (Tom)
修复在一个表达式之内比较的 subselect (Jan)
修复返回行时错误报告的处理 (Tom)
修复参考数组类型的问题 (Tom,Jan)
阻止 UPDATE SET oid (Jan)
修复 pg_dump 使 -t 选项可以处理大小写敏感的表名
修复特殊情况下的 GROUP BY (Tom, Jan)
修复失败查询中的内存泄露 (Tom)
DEFAULT 现在支持混合情况的标识符 (Tom)
修复 DROP/RENAME 表、索引的多节的使用 (Ole Gjerde)
使用 -o 和 -d 选项禁用 pg_dump 的使用 (Bruce)
允许 pg_dump 适当的转储组权限 (Bruce)
修复 INSERT INTO table SELECT * FROM table2 中的 GROUP BY (Jan)
修复视图中的计算 (Jan)
修复数组索引上的聚集 (Tom)
修复在需要太多引号的值内 DEFAULT 处理单引号
修复非超级用户导入/导出大对象的安全问题 (Tom)
回滚创建表被适当的清理的事务 (Tom)
修复允许大表和字段名产生适当的序列名 (Tom)

增强
------------
添加 "vacuumdb" 工具
通过更好的分配存储器加速 libpq (Tom)
EXPLAIN 所有使用的索引 (Tom)
```

```
实现 CASE, COALESCE, NULLIF 表达式 (Thomas)
新增 pg_dump 表输出格式 (Constantin)
添加字符串 min()/max() 函数 (Thomas)
扩展新的类型强制转换技术到聚集中 (Thomas)
新增 moddatetime 贡献包 (Terry)
升级到 pgaccess 0.96 (Constantin)
为单字节 "char" 类型添加例程 (Thomas)
改善 substr() 函数 (Thomas)
改善多字节处理 (Tatsuo)
多版本并发控制/MVCC (Vadim)
新增序列化模式 (Vadim)
修复超过 2gigs 的表 (Peter)
新增 SET TRANSACTION ISOLATION LEVEL (Vadim)
新增 LOCK TABLE IN ... MODE(Vadim)
升级 ODBC 驱动程序 (Byron)
新增 NUMERIC 数据类型 (Jan)
新增 SELECT FOR UPDATE(Vadim)
处理输入值的 "NaN" 和 "Infinity" (Jan)
改善日期/年的处理 (Thomas)
改善后端连接的处理 (Magnus)
为大文件新增 ELOG_TIMESTAMPS 和 USE_SYSLOG 选项 (Massimo)
新增 TCL_ARRAYS 选项 (Massimo)
新增 INTERSECT 和 EXCEPT (Stefan)
为主键跟踪新增 pg_index.indisprimary (D'Arcy)
新增 pg_dump 选项以允许在创建之前删除表 (Brook)
加速行输出例程 (Tom)
新增 READ COMMITTED 隔离级别 (Vadim)
新增 TEMP 表/索引 (Bruce)
如果结果已经排序那么阻止排序 (Jan)
新增内存分配优化 (Jan)
允许 psql 做 \p\g (Bruce)
允许多重的规则动作 (Jan)
添加了 LIMIT/OFFSET 功能 (Jan)
改善了连接大量的表时的优化器 (Bruce)
新增简介到 SQL，来自 S. Simkovics 的硕士论文 (Stefan, Thomas)
新增简介到后端处理，来自 S. Simkovics 的硕士论文 (Stefan)
改进了 int8 支持 (Ryan Bradetich, Thomas, Tom)
新增转换 int8 和 text/varchar 类型的例程 (Thomas)
在连接了 meta-tables 的地方新增了严密的规划 (Bruce)
缺省启用右边的查询 (Bruce)
允许在配置时设置后端可靠的最大数目(--with-maxbackends and postmaster switch (-N backends)) (Tom
GEQO 因为优化器加速现在缺省为 10 个表 (Tom)
为了 MS-SQL 可移植性允许 NULL=Var (Michael, Bruce)
修改贡献包 check_primary_key()，因此都 "自动的" 或 "依赖的" (Anand)
允许 psql 在一个视图上 \d 显示查询 (Ryan)
为 LIKE 加速 (Bruce)
Ecpg 修复/特性，查阅 src/interfaces/ecpg/ChangeLog 文件 (Michael)
JDBC 修复/特性，查阅 src/interfaces/jdbc/CHANGELOG (Peter)
使 % 运算符和 / 一样有优先级 (Bruce)
添加新的 postgres -O 选项，允许改变系统表结构 (Bruce)
更新 contrib/pginterface/findoidjoins 脚本 (Tom)
主要的加速在于 vacuum 的删除带有索引的行 (Vadim)
允许非 SQL 函数运行基于参数的不同版本 (Tom)
添加 -E 选项显示由 \dt 和朋友们发送的实际查询 (Masaaki Sakaida)
为 psql 在启动标题中添加版本号 (Masaaki Sakaida)
新增 contrib/vacuumlo 删除没有引用的大对象 (Peter)
新增初始化表的大小，这样非真空的表执行的更好 (Tom)
改善拒绝连接时的错误信息 (Tom)
支持数组的 char() 和 varchar() 字段(Massimo)
彻底检修哈希码以增强可靠性和性能 (Tom)
升级到 PyGreSQL 2.4 (D'Arcy)
改变调试选项 so -d4 和 -d5 产生不同的节点显示 (Jan)
新增 pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
系统表访问的更好的优化统计 (Tom)
非缺省块大小的更好的处理 (Massimo)
改进 GEQO 优化器内存消耗 (Tom)
UNION 现在支持不在目标列表中的字段的 ORDER BY (Jan)
改进了主要的 libpq++ (Vince Vielhaber)
pg_dump 现在使用 -z(ACL's) 作为缺省 (Bruce)
后端缓存、内存加速 (Tom)
让 pg_dump 在一个快照事务中做任何事情 (Vadim)
修复大对象内存泄露，修复 pg_dumping (Tom)
```

```
INET 类型现在关心网络掩码的比较
使 VACUUM ANALYZE 只使用一个读锁 (Vadim)
允许 UNIONS 上的 VIEW (Jan)
pg_dump 现在可以在活动的数据库上生成一致的快照 (Vadim)

 源代码树的变化
-------------------
改进端口匹配 (Tom)
SunOS 的可靠性修复
添加 Windows NT 后端端口和启用动态加载 (Magnus 和 Daniel Horak)
新增端口到 Cobalt Qube(Mips) 运行 Linux (Tatsuo)
到 NetBSD/m68k 的端口 (Mr. Mutsuki Nakajima)
到 NetBSD/sun3 的端口 (Mr. Mutsuki Nakajima)
到 NetBSD/macppc 的端口 (Toshimi Aoki)
修复 tcl/tk 配置 (Vince)
为规则查询删除 CURRENT 关键字 (Jan)
NT 动态加载现在可以运行了 (Daniel Horak)
添加 ARM32 支持 (Andrew McMurry)
对 HP-UX 11 和 UnixWare 的更好的支持
改进文件处理使其更统一，阻止文件描述符泄露 (Tom)
新增 plpgsql 的安装命令 (Jan)
```

E.227. 版本 6.4.2

> 发布日期: 1998-12-20

6.4.1 版本的打包不正确。这个版本也有一些 bug 修复。

E.227.1. 迁移到版本 6.4.2

运行 6.4.* 的用户不需要转储/恢复。

E.227.2. 修改列表

修复了某些平台上日期时间常量的问题 (Thomas)

E.228. 版本 6.4.1

> 发布日期: 1998-12-18

这基本上是6.4的一个净化版本。我们修复了一些6.4的用户报告的问题。

E.228.1. 迁移到版本 6.4.1

运行 6.4 的用户不需要转储/恢复。

E.228.2. 修改列表

```
添加 pg_dump -N 标识强制双引号包围标识符。这是缺省 (Thomas)
修复 where 子句中的 NOT 导致崩溃 (Bruce)
EXPLAIN VERBOSE 内核转储修复 （Vadim）
修复 Linux 上的共享库问题
修复表存在的测试, 允许在表名中存在混合大小写和空格 (Thomas)
修复了两个 pg_dump bug
配置更好的匹配 template/.similar 条目 (Tom)
更改内建函数名 SPI_* 为 spi_*
OR WHERE 子句修复 （Vadim）
修复了混合大小写的表名 （Billy）
contrib/linux/postgres.init.csh/sh 修复 (Thomas)
libpq 内存溢出修复
SunOS 修复 （Tom）
改变 exp() 的行为以在下溢时产生错误 (Thomas)
修复 pg_dump 内存泄露, 继承约束, 布局改变
升级 pgaccess 到 0.93
为 64 位平台修复原型
多字节修复 (Tatsuo)
新增 ecpg 手册页
修复内存溢出 （Tatsuo）
修复 lo_import() 崩溃 (Bruce)
安装程序更好的搜索 (Tom)
时区修复 (Tom)
HP-UX 修复 (Tom)
使用隐式的类型转换以匹配 DEFAULT 值 (Thomas)
添加例程以帮助单字节（内部的）字符类型 (Thomas)
为 Windows 修复编辑 libpq (Magnus)
升级到 PyGreSQL 2.2 (D'Arcy)
```

E.229. 版本 6.4

> 发布日期: 1998-10-30

这个版本有许多新特性和改进。感谢开发者和维护者，与前一个版本相比， 几乎系统的每个方面都得到了重视。下面是一个不完全的概要：

- 感谢 Jan Wieck 为重写规则系统添加的新的代码，视图和规则现在可以工作了。 他就此还为程序员手册写了一章。
- 与他在上一版本提供的最初的 PL/pgTCL 过程语言一起， Jan 还提供了第二种过程语言，PL/pgSQL。
- 我们有了可选的多字节字符集支持，是 Tatsuo Iishi 提供的，用以完善我们现存的本地支持。
- 感谢 Tom Lane 清理了 Client/server（客户/服务器）通讯部分，对异步消息和中断有了更好的支持.
- 分析器现在会进行自动的类型转换，以与现有的操作符和函数的参数相匹配， 以及与目标字段的字段（类型）和表达式匹配。这是通过通用的机制实现的， 该机制支持 PostgreSQL 的类型扩展特性。 在用户手册中有一章介绍这个主题。
- 新增了三种数据类型。其中两种类型 `inet` 和 `cidr` ， 支持各种各样的IP网络，子网和机器寻址形式。而且现在在某些平台上有8字节整数支持了。 请参阅用户手册中的数据类型章节获取详细信息。第四种类型， `serial` ，现在被分析器当作 `int4` 类型，序列号，和唯一索引的混合物来支持。
- 增加了更多 SQL92 兼容的语法特性，包括 `INSERT DEFAULT VALUES` 。
- 自动配置和安装系统受到了一定的重视，现在在更多平台上应该比以前更坚固， 稳定了。

E.229.1. 迁移到版本 6.4

要想从以前的 PostgreSQL 版本迁移到新的版本， 你需要用 pg_dump 或 pg_dumpall 进行一个转储/恢复的操作。

E.229.2. 修改列表

```
Bug 修复
---------
修复 PQsetdb/PQfinish 中微小的内存泄露 (Bryan)
```

```
使用 char/varchar 删除 char2-16 数据类型 (Darren)
Pqfn 不处理 NOTICE 信息 (Anders)
减少了许多后端自旋锁的 busywaiting 开销 (dg)
自旋锁检测卡住 (dg)
修复 "ISO-style" 时间间隔解码和编码 (Thomas)
修复在事务回滚之后删除表的问题 (Vadim)
更改错误信息和删除非功能性的升级信息 (Vadim)
修复 COPY 数组检测
修复 SELECT 1 UNION SELECT NULL
修复大对象调用中的缓存泄露 (Pascal)
改变所有者从 oid 到 int4 类型 (Bruce)
修复 oracle 兼容性函数 btrim() ltrim() 和 rtrim() 中的一个bug
修复共享的失效缓存溢出 (Massimo)
阻止文件描述符泄漏到已失败的 COPY (Bruce)
修复 libpgtcl 的 pg_select 中的内存泄露 (Constantin)
修复用户名/口令超过8个字符的问题 (Tom)
修复在后端处理异步的 NOTIFY 问题 (Tom)
修复许多损坏了的系统表条目 (Tom)

增强
------------
升级 ecpg 和 ecpglib，参阅 src/interfaces/ecpc/ChangeLog(Michael)
显示在一个 EXPLAIN 中使用的索引 (Zeugswetter)
EXPLAIN 调用规则系统并显示重写查询的规划 (Jan)
通过配置，多字节意识到许多数据类型和函数 (Tatsuo)
新增 configure --with-mb 选项 (Tatsuo)
新增 initdb --pgencoding 选项 (Tatsuo)
新增 createdb -E multibyte 选项 (Tatsuo)
Select version(); 现在返回 PostgreSQL 版本 (Jeroen)
libpq 现在允许异步的客户端 (Tom)
允许取消客户端的后端查询 (Tom)
psql 现在用 Control-C 取消查询 (Tom)
libpq 用户不需要发出虚假的查询来获取 NOTIFY 信息  (Tom)
NOTIFY 现在可以发送发送者的 PID，所以你可以知道它是否是你自己的 (Tom)
PGresult 结构现在包括关联的错误信息，如果有的话 (Tom)
定义 "tz_hour" 和 "tz_minute" 参数到 date_part()(Thomas)
添加 varchar 和 bpchar 之间相互转换的例程 (Thomas)
添加允许 varchar 和 bpchar 的大小到目标字段的例程 (Thomas)
添加位标记以在数据检索中支持时区小时和分钟 (Thomas)
允许更多的变化在浮点数上 (如 ".1", "1e6") (Thomas)
修复带有前导空格的一元符号语法分析 (Thomas)
实现了 TIMEZONE_HOUR, TIMEZONE_MINUTE 每 SQL92 规格 (Thomas)
检查并适当的忽略 FOREIGN KEY 字段约束 (Thomas)
定义 USER 作为 CURRENT_USER 的同义词每 SQL92 规格 (Thomas)
启用 HAVING 子句，但是在别处还没有修复
使 "char" 类型是 "char(1)" 的同义词（实际上是作为 bpchar 来实现） (Thomas)
如果为 DEFAULT 子句处理指定则保存字符串类型 (Thomas)
强制操作包括不同的数据类型 (Thomas)
允许一些索引用于不同类型的字段 (Thomas)
为自动类型转换添加功能 (Thomas)
清理大对象，因此文件打开时被截断 (Peter)
Readline 清理(Tom)
允许 psql  \f \ 将空格作为分隔符 (Bruce)
为列字段长度传送 pg_attribute.atttypmod 到前端 (Tom,Bruce)
Msql 兼容库在 /contrib 中 (Aldrin)
删除 ORDER/GROUP BY 子句标识符被包含在目标列表中的要求 (David)
转换字段以匹配 UNION 子句中的字段 (Thomas)
删除 fork()/exec() 只进行 fork() (Bruce)
Jdbc 清理 (Peter)
在 ps 命令行上显示后端状态 (只在某些平台上工作) (Bruce)
Pg_hba.conf 现在在数据库字段中有一个 sameuser 选项
使 lo_unlink 接受 oid 参数，而不是 int4
为不能处理我们的宏命令的编译器新增 DISABLE_COMPLEX_MACRO (Bruce)
Libpgtcl 现在把 NOTIFY 当做一个 Tcl 事件处理，不需要发送脏查询 (Tom)
libpgtcl 清理(Tom)
添加 -error 选项到 libpgtcl 的 pg_result 命令(Tom)
新增本地路径，参阅 docs/README/locale (Oleg)
修复 pg_dump，因此 CONSTRAINT 和 CHECK 语法是不正确的 (ccb)
为除去孤立的大对象新增 contrib/lo 代码 (Peter)
为多字节特性新增 psql 命令 "SET CLIENT_ENCODING TO 'encoding'"，参阅 /doc/README.mb (Tatsuo)
contrib/noupdate 代码以撤销在一个字段上的更新许可
libpq 现在可以在 Windows 上编译 (Magnus)
```

```
在 libpq 里添加 PQsetdbLogin()
新增8字节整数类型，为了 OS 支持由配置校对 (Thomas)
对引用表/字段名有了更好的支持 (Thomas)
pg_dump 中的表和字段名由双引号包围 (Thomas)
PQreset() 现在带有口令工作 (Tom)
处理 GROUP BY 目标列表字段编号超出范围的情况 (David)
允许在子查询中 UNION
添加 auto-size 到屏幕的 \d? 命令 (Bruce)
使用 UNION 显示一个查询中所有 \d? 的结果 (Bruce)
添加 \d? 字段搜索特性 (Bruce)
Pg_dump 发出少数 \connect  请求 (Tom)
使 pg_dump -z 标识工作的更好，在手册页中记录它 (Tom)
添加全力支持子查询和联合的 HAVING 子句 (Stephan)
全文本搜索例程在 contrib/fulltextindex 中 (Maarten)
事务 id 现在存储在共享内存中 (Vadim)
新增发出 COPY 命令时的 PGCLIENTENCODING (Tatsuo)
支持 SQL92 语法 "SET NAMES" (Tatsuo)
支持 LATIN2-5 (Tatsuo)
添加 UNICODE 回归测试案例 (Tatsuo)
锁住管理者清理，为 LLL 新增锁模式 (Vadim)
允许索引和 OR 子句一起使用 (Bruce)
允许 "SELECT NULL ORDER BY 1;"
Explain VERBOSE 打印规划，现在倾向于打印规划到主进程日志文件 (Bruce)
添加索引显示到 \d 命令 (Bruce)
允许在函数上 GROUP BY (Bruce)
为大对象新增 pg_class.relkind (Bruce)
新的方式发送 libpq NOTICE 信息到不同的位置 (Tom)
新增 \w 写入命令到 psql (Bruce)
新增 /contrib/findoidjoins 扫描 oid 字段以找出连接关系 (Bruce)
当为包含一个常量的约束子句检查有效索引时允许被认为是二进制兼容的索引 (Thomas)
在 /contrib/isbn_issn 中新增 ISBN/ISSN 代码
允许 NOT LIKE, IN, NOT IN, BETWEEN, 和 NOT BETWEEN 约束 (Thomas)
新的重写系统修复了许多规则和视图的问题 (Jan)
    * 关系上的规则可以工作了
    * 在 insert/update/delete 上的事件限制可以工作了
    * 新增 OLD 变量引用 CURRENT，CURRENT 将在将来删除
    * 更新规则可以在规则限制/动作中引用 NEW 和 OLD
    * 在视图上的 Insert/update/delete 可以工作了
    * 现在支持多规则动作了，由圆括号包围
    * 固定用户可以在他们有 RULE 许可的表上创建视图/规则
    * 规则和视图继承创建者的权限
    * 没有规则是字段级别的
    * 没有 UPDATE NEW/OLD 的规则
    * 新增 pg_tables, pg_indexes, pg_rules 和 pg_views 系统视图
    * 在 SELECT 规则上只有一个单一动作
    * 完全重写改革，可能是为了6.5
    * 处理子查询
    * 处理视图上的聚集
    * 处理 insert into select from view 工作了
系统索引现在是多键的 (Bruce)
删除了 Oidint2, oidint4, 和 oidname 类型 (Bruce)
为更多系统表查找使用系统缓存 (Bruce)
在 backend/pl 中新增后端编程语言 PL/pgSQL (Jan)
新增 SERIAL 数据类型，自动创建序列/索引 (Thomas)
启用不带有重新编译的维护检查 (Massimo)
用户锁增强 (Massimo)
新增 setval() 命令设置序列值 (Massimo)
如果没有主进程正在运行，那么在启动时自动删除 unix 套接字文件 (Massimo)
有条件的跟踪包裹 (Massimo)
新增 UNLISTEN 命令 (Massimo)
psql 和 libpq 现在在 Windows 下使用 win32.mak 编译 (Magnus)
Lo_read 不再存储尾随的 NULL (Bruce)
标识符现在在内部截断为31为字符 (Bruce)
Createuser 选项现在在命令行中可用
为64为整数支持的编码添加了，配置测试，int8类型 (Thomas)
阻止来自失败 COPY 的文件描述符页 (Bruce)
新增 pg_upgrade 命令 (Bruce)
升级了 /contrib 目录 (Massimo)
新增 CREATE TABLE DEFAULT VALUES 声明可用 (Thomas)
新增 INSERT INTO TABLE DEFAULT VALUES 声明可用 (Thomas)
新增 DECLARE 和 FETCH 特性 (Thomas)
现在不再输出 libpq 的内部构件 (Tom)
```

```
允许多达8个键字索引 (Bruce)
删除 ARCHIVE 关键字，不再使用它了 (Thomas)
pg_dump -n 标记抑制包围标识符的引号
为视图禁用系统字段 (Jan)
为网络地址新增 INET 和 CIDR 类型 (TomH, Paul)
在 psql 输出中不再有双引号
pg_dump 现在可以转储视图了 (Terry)
新增 SET QUERY_LIMIT (Tatsuo,Jan)

源代码树变化
-------------------
/contrib 清理 (Jun)
为每行内联一些小的函数调用 (Bruce)
Alpha/linux 修复
HP-UX 清理 (Tom)
多字节回归测试 (Soonmyung.)
从配置中删除 --disabled 选项
定义 PGDOC 以在缺省情况下使用 POSTGRESDIR
使回归可选
删除 pgindent 中多余的花括号代码 (Bruce)
添加 bsdi 共享库支持 (Bruce)
新增 --without-CXX 支持配置选项 (Brook)
新增 FAQ_CVS
更新 tools/backend 中的后端流程图 (Bruce)
将 atttypmod 从 int16 更改为 int32 (Bruce, Tom)
修复平台上没有 Getrusage() 的问题 (Tom)
添加 PQconnectdb, PGUSER, PGPASSWORD 到libpq手册页
NS32K 平台修复 (Phil Nelson, John Buller)
SCO 7/UnixWare 2.x 修复(Billy,others)
Sparc/Solaris 2.5 修复(Ryan)
Pgbuiltin.3 已经废弃了，移动到了 doc 文件中 (Thomas)
更多的文档 (Thomas)
Nextstep 支持 (Jacek)
Aix 支持 (David)
pginterface 手册页 (Bruce)
共享库都有版本号
合并所有 OS 特定的共享库定义到一个文件
更智能的 TCL/TK  配置检测 (Billy)
更智能的 perl 配置 (Brook)
如果没有发现安装脚本，那么配置使用提供的安装 sh (Tom)
为共享库配置新增 Makefile.shlib (Tom)
```

E.230. 版本 6.3.2

> 发布日期: 1998-04-07

这是一个 6.3.x 的bug修复版本。请参阅版本6.3的版本信息获取新特性的更完整的概述信息。

概要：

- 修复了对一些平台（包括 Linux）的自动配置的支持， 在v6.3.1中由于一些小错误带进来了这些破损。
- 正确地处理在 BETWEEN 和 LIKE 子句左边的函数调用。

对运行在6.3或6.3.1上的数据库不需要进行转储/恢复工作。 所要做的所有工作是 `make distclean` ， `make` ， 和 `make install` 。最后一步应该在 postmaster 停止运行后进行。 你应该用 PostgreSQL 库对所有用户应用进行重新链接.

对于从v6.3以前的版本升级，请参考v6.3的安装和移植指导。

E.230.1. 修改列表

```
改善了 tcl/tk 的配置检测 (Brook Milligan, Alvin)
改进了手册页 (Bruce)
修复了 BETWEEN 和 LIKE (Thomas)
修复了 pg_dump 使用的 psql \connect (Oliver Elphick)
新增 odbc 驱动器
pgaccess, version 0.86
删除了 qsort，现在使用 libc 版本，清理 (Jeroen)
修复了缓存溢出检测 (Maurice Gittens)
修复了在 libpgtcl 中的缓存溢出 (Randy Kunkee)
修复了带有 DISTINCT 或 ORDER BY 的 UNION (Bruce)
gettimeofday 配置检测 (Doug Winterburn)
修复了 "indexes not used" bug(Vadim)
文档添加 (Thomas)
修复了后端内存泄露 (Bruce)
libreadline 清理 (Erwan MAS)
删除 DISTDIR (Bruce)
Makefile 依赖清理 (Jeroen van Vianen)
ASSERT 修复 (Bruce)
```

E.231. 版本 6.3.1

> 发布日期: 1998-03-23

摘要：

- 对多字节字符集的附加支持。
- 修复了混合字节序的服务器和客户端环境的支持。
- 对允许的 SQL 语法进行了小的升级。
- 改善了安装过程的配置自动检测功能。

对那些运行着6.3版本的应用不需要进行转储/恢复操作。升级工作只需要 `make distclean`， `make`，和 `make install`。 最后一步应该在 postmaster 没有运行时进行。升级后你应该对所有使用 PostgreSQL 库的用户应用重新进行连接.

对于从v6.3以前的版本升级，请参考v6.3的安装和移植指导。

E.231.1. 修改列表

```
ecpg 清理/修复，现在的版本是 1.1 (Michael Meskes)
pg_user 清理 (Bruce)
pg_dump 和 tclsh 的大对象修复 (alvin)
修复多个临近的下划线的 LIKE
修复重新定义内建函数 (Thomas)
ultrix4 清理
升级到 pg_access 0.83
更新了 CLUSTER 手册页
多字节字符设置支持，参阅 doc/README.mb (Tatsuo)
configure --with-pgport 修复
pg_ident 修复
为后端通信修复 big-endian (Kataoka)
SUBSTR() 和 substring() 修复 (Jan)
几个 jdbc 修复(Peter)
libpgtcl 改善，参阅 libptcl/README (Randy Kunkee)
修复了 "Datasize = 0" 错误 (Vadim)
阻止 \do 打包 (Bruce)
删除重复的俄语字符集条目
Sunos4 清理
在 LOCK 和 SELECT INTO 中允许 TABLE 关键字可选 (Thomas)
CREATE SEQUENCE 选项允许负整数 (Thomas)
作为一个允许的字段标识符添加 "PASSWORD" (Thomas)
为 UNION 目标字段添加检查 (Bruce)
修复了 Alpha 端口 (Dwayne Bailey)
修复了文本数组包含引号的问题 (Doug Gibson)
Solaris 编译修复 (Albert Chin-A-Young)
更好的识别出 tcl 和 tk 库和包含 (Bruce)
```

E.232. 版本 6.3

> 发布日期: 1998-03-01

在这个版本中有许多新的特性和改进。下面是一个简短的不完整的概要：

- 增加了许多新的 SQL 特性，包括完整的 SQL92 子查询功能（除目标列子查询外所有的东西都实现了）。
- 支持客户端环境变量声明时区和日期风格。
- 用于 客户端/服务器 联接的套接字（Socket）接口。现在这个是缺省的值， 所以你可能需要带 `-i` 参数运行 postmaster。
- 更完善的口令认证机制。修改了缺省的表权限。
- 删除了老式的时间跟踪（*time travel*）特性。改善了性能。

> **Note:** Bruce Momjian 写下了下面的介绍新版本的文字。

我在这里想提一些关于 6.3 的事情。这些都是一些无法在一句话中描述的比较大的课题。 所以仍需要你回过头去看看修改的详细列表。

首先，我们现在有了子查询。既然我们已经拥有子查询了，我要很客观的说， 如果没有子查询，SQL 就是一种作用非常有限的语言。子查询是一种很重要的特性， 你应该复查一下你的代码，以便找出子查询能给你提供更好解决方法的地方。 我相信你会发现子查询可以应用在比你想象得到多得多的地方。 Vadim 把我们放在了一个拥有完整功能的带子查询功能的巨大的 SQL前景上面。 你不能应用子查询的唯一方面是目标列。

第二，6.3 使用 unix 域套接字作为缺省而不是 TCP/IP。要允许从其他机器来的连接， 你必须使用新的 postmaster 的 -i 选项，当然你还要编辑 `pg_hba.conf` 。 同样， `pg_hba.conf` 的格式也因此而改变了。

第三，现在使用 `char()` 字段将比使用 `varchar()` 或 `text` 时访问的更快。 具体地说，在对第一次 `text` 和 `varchar()` 的字段进行访问之后再对同类型字段访问将有一些访问延迟。 `char()` 原先也有这种访问延迟， 但现在已经没有了。这或许意味着你重新设计你的一些表，尤其是那些你已经定义为 `varchar()` 或 `text` 的短的字符型字段。这个和其他的一些修改使 6.3 比早期的版本有更快的速度。

我们现在有了独立于任何 Unix 文件的可定义的口令。现在有了新的 SQL USER 命令。 参阅管理员手册获取更多信息。同时我们还有了一个新的表，pg_shadow， 用来存放用户信息和用户口令，并且缺省时只有 postgres 超级用户对其有 （SELECT）查询权限。pg_user 现在是 pg_shadow 的一个视图，并且可以被公众（PUBLIC）读取（SELECT）。 你可以在你的应用里继续使用 pg_user 而不需做任何更改。

用户创建的表缺省时不再被公众（PUBLIC）拥有读取（SELECT）权限。这么做是因为ANSI 标准要求这样做。 你当然可以在创建表后用 GRANT 赋予他人任何你希望的权限。系统表仍然可以被公众（PUBLIC）读取（SELECT）。

我们仍然有实时的死锁检测代码。超时时间不超过60秒。并且新的锁定代码实现了更好的FIFO （先入先出），所以在重负荷时对资源的需求会轻一些。

在以前的版本中我们听到了许多关于缺乏文档的抱怨。Thomas 在这个版本中努力增加了许多新的手册。 请查阅 doc/ 目录.

出于性能考虑，时间跟踪特性取消了，但是可以使用触发器实现（参阅 `pgsql/contrib/spi/README` ）。 请使用新的 \d 命令查询关于类型，操作符等的信息。同时，视图拥有了他们自己的权限，而不是以它们依赖的表为基础。 所以它们的权限应该独立的设置。请检查 `/pgsql/interfaces` 文件获取更多与 PostgreSQL 对话的方法。

这是第一个需要向已有的用户进行说明的版本。不管从哪个角度来说，我们都需要做这样的说明， 因为新版本取消了许多限制，因而许多原先人们需要做的许多防范性工作都不再需要了.

E.232.1. 迁移到版本 6.3

任何希望从以前的 PostgreSQL 版本移植到新版本的数据库都需要用 pg_dump 或 pg_dumpall 进行一次转储/恢复工作。

E.232.2. 修改列表

```
Bug 修复
---------
修复 MOVE 实现毁坏的二进制游标 (Vadim)
修复 tcl 库崩溃 (Jan)
修复数组处理，来自 Gerhard Hintermayer
修复 acl 错误，删除重复的 pqtrace (Bruce)
为空文件修复 psql \e (Bruce)
修复 varchar() 字段上的 textcat (Bruce)
修复 DBT Sendproc (Zeugswetter Andres)
修复 vacuum 分析语法问题 (Bruce)
修复国际标识符 (Tatsuo)
修复在继承表上的聚集 (Bruce)
为超出范围的数据修复 substr()
修复 select 1=1 or 2=2, select 1=1 and 2=2, 和 select sum(2+2) (Bruce)
修复 notty 输出显示状态的结果。-q 选项仍然是将它关闭 (Bruce)
修复带有视图和多行表和 sum(3) 的 count(*), aggs (Bruce)
修复群集 (Bruce)
几次修复了 PQtrace start/stop (Bruce)
修复各种各样的锁问题，像较新的锁等待在较旧的锁等待之前得到锁，
        如果一个写入正在等待一个锁使不可读的人员不共享锁，
    等待写入的没有获得比等待读取的更高的优先权 (Bruce)
修复了从外部文件执行查询时 psql 中的崩溃 (James)
修复了有多个命令列切第一个是 NULL 值的问题 (Jeroen)
为 float8 和 int4 使用正确的哈希表支持函数 (Thomas)
重新在 CREATE OPERATOR 语句中启用 JOIN= 选项 (Thomas)
改变布尔操作符的优先级以匹配预期行为 (Thomas)
在超大的整数上产生 elog(ERROR) (Bruce)
```

```
在约束子句中允许多个参数的函数 (Thomas)
检查布尔输入文本 'true','false','yes','no','1','0' 并在不能识别时抛出 elog(ERROR) (Thomas)
主要的大对象修复
修复 GROUP BY 显示重复的问题 (Vadim)
修复 MergeJoin 中的索引扫描 (Vadim)

增强
------------
带有 EXISTS, IN, ALL, ANY 关键字的子查询 (Vadim, Bruce, Thomas)
新增用户手册 (Thomas, others)
通过嵌入一些频繁调用的函数加速
真实的死锁检测，不再超时 (Bruce)
添加 SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER (Thomas
修改约束语法使其 SQL92 兼容 (Thomas)
使用索引实现 SQL92 PRIMARY KEY 和 UNIQUE 子句 (Thomas)
为 FOREIGN KEY 识别出 SQL92 语法。抛出 elog 通知 (Thomas)
允许 NOT NULL UNIQUE 子句(每个语句在之前单独的被允许)(Thomas)
允许非常量 PostgreSQ 风格的转换 ("::") (Thomas)
添加对 SQL3 TRUE 和 FALSE 布尔常量的支持 (Thomas)
支持 SQL92 语法 IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE (Thomas)
允许布尔字面值是较短的字符串 (如 "t", "tr", "tru") (Thomas)
允许 SQL92 分隔标识符 (Thomas)
实现 SQL92 二进制和十六进制字符串编码 (b'10' 和 x'1F') (Thomas)
支持 SQL92 语法文字字符串的类型强制转换(如 "DATETIME 'now'") (Thomas)
添加 int2, int4, 和 OID 类型和文本类型之间的转换 (Thomas)
在建立索引时使用共享锁 (Vadim)
在一个用户查询完成之后，在一个事务块内部给该查询分配空闲存储，截断为 <= 6.2.1 (Vadim)
新增 SQL 语句 CREATE PROCEDURAL LANGUAGE (Jan)
新增 PostgreSQL 过程语言 (PL) 后端接口 (Jan)
重命名 pg_dump -H 选项为 -h (Bruce)
添加 Java 支持口令，European 日期 (Peter)
使用索引 LIKE 和 ~, !~ 操作符 (Bruce)
为 datetime 和 timespan 添加哈希函数 (Thomas)
删除了 Time Travel (Vadim, Bruce)
为 \d 和 \z 添加分页，并且修复了 \i (Bruce)
添加 Unix 域套接字支持到后端和前端库 (Goran)
实现了 CREATE DATABASE/WITH LOCATION 和 initlocation 工具 (Thomas)
允许更多 SQL92 和/或 PostgreSQL 保留字作为字段标识符 (Thomas)
增大对 SQL92 SET TIME ZONE... 的支持 (Thomas)
SET/SHOW/RESET TIME ZONE 使用 TZ 后端环境变量 (Thomas)
实现了 SET keyword = DEFAULT 和 SET TIME ZONE DEFAULT (Thomas)
启用 SET TIME ZONE 使用 TZ 环境变量 (Thomas)
添加 PGDATESTYLE 环境变量到前端和后端初始化 (Thomas)
添加 PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO 前端库初始化环境变量 (Thomas)
回归测试时区自动设置 "setenv PGTZ PST8PDT" (Thomas)
为表、字段、操作符、类型和聚集的信息添加 pg_description 表 (Bruce)
增加系统表/索引名上的 16 字符限制到 32 个字符 (Bruce)
重命名系统索引 (Bruce)
添加 'GERMAN' 选项到 SET DATESTYLE (Thomas)
定义一个带有 "hh:mm:ss" 字段的  "ISO-style" 时间间隔输出格式 (Thomas)
允许时间增量的小数值 (如 '2.5 days') (Thomas)
为时间增量更细心的验证数值输入 (Thomas)
实现了一年中的天数作为 date_part() 的可能输入 (Thomas)
定义 timespan_finite() 和 text_timespan() 函数 (Thomas)
删除存档的东西 (Bruce)
允许 pg_password 认证数据库从系统口令文件中分离出来 (Todd)
转储 ACLs, GRANT, REVOKE 权限 (Matt)
定义 text, varchar, 和 bpchar 字符串长度函数 (Thomas)
修复继承的查询处理，和开销计算 (Bruce)
实现了 CREATE TABLE/AS SELECT (作为 SELECT/INTO 的替换)(Thomas)
在约束中允许 NOT, IS NULL, IS NOT NULL (Thomas)
为 SELECT 实现了 UNION (Bruce)
添加 UNION, GROUP, DISTINCT 到 INSERT (Bruce)
varchar() 存储只需要磁盘上的字节 (Bruce)
修复了 BLOBs (Peter)
JDBC... 的 Mega-Patch 参阅 README_6.3 的修改列表 (Peter)
删除了 PQconnectdb() 中不使用的 "option"
新增 LOCK 命令和描述死锁的锁手册页(Bruce)
添加了新的 psql \da, \dd, \df, \do, \dS, 和 \dT 命令(Bruce)
增强了 psql \z 以显示序列 (Bruce)
在 psql \d 的表中显示 NOT NULL 和 DEFAULT (Bruce)
新增 psql .psqlrc 文件启动 (Andrew)
```

```
在 contrib/linux 中修改样本启动脚本以显示系统日志 (Thomas)
在 contrib/ip_and_mac 中为 IP 和 MAC 地址新增类型 (TomH)
contrib/unixdate 中 Unix 系统时间和 日期/时间 类型的转换 (Thomas)
贡献版的更新 (Massimo)
添加 Unix 套接字支持到 DBD::Pg (Goran)
新增 python 接口 (PyGreSQL 2.0)(D'Arcy)
新的前端/后端控制有一个版本号，网络字节顺序 (Phil)
pg_hba.conf 中的安全特征加强了也记录了，还有许多清理 (Phil)
CHAR() 现在比 VARCHAR() 或 TEXT 访问更快
ecpg 嵌入了 SQL 预处理器
减少系统字段开销 (Vadmin)
删除 pg_time 表 (Vadim)
添加 pg_type 属性以确定需要长度的类型 (bpchar, varchar)
当 COPY 命令失败时添加违规行的报告
允许 VIEW 权限与基础表的设置分离开来。为了安全，根据情况在视图上使用 GRANT/REVOKE (Jan)
表现在没有缺省的 GRANT SELECT TO PUBLIC。你必须明确的赋予这样的权限。
清理 教程示例 (Darren)

源代码树的变化
-------------------
添加新的 html 开发工具，和在 /tools/backend 中添加流程图
修复 SCO 编译
Robert Gillies 层云计算机接口
为 BSD44_derived shlib 添加支持  & i386_solaris
使配置更加自动化 (Brook)
添加检查回归测试结果的脚本
分解解析器函数为更小的文件，集合到一起 (Bruce)
重命名 heap_create 为 heap_create_and_catalog，重命名 heap_creatr 为 heap_create() (Bruce)
为锁定修补 Sparc/Linux (TomS)
删除 PORTNAME 并整理端口特定的东西 (Marc)
添加优化器 README 文件 (Bruce)
删除优化器中的一些递归并清理此处的一些代码 (Bruce)
修复 NetBSD 锁定 (Henry)
修复 libptcl 制造 (Tatsuo)
AIX 修补 (Darren)
更改 IS TRUE, IS FALSE, ... 到表达式，使用 "=" 然不是函数调用 istrue() 或 isfalse() 来允许优化 (
修复各种 NetBSD/Sparc 相关 (TomH)
Alpha linux 锁定 (Travis,Ryan)
更改 elog(WARN) 为 elog(ERROR) (Bruce)
FreeBSD 的 FAQ (Marc)
引入 PostODBC 源代码树作为我们的标准发布的一部分 (Marc)
HP/UX 10 vs 9 的小幅度修补(Stan)
为 type-specific 信息新增 pg_attribute.atttypmod，就像 varchar 长度 (Bruce)
UnixWare 补丁 (Billy)
为自旋锁 asm 新增 i386 'lock' (Billy)
多路复用后端的支持已经删除了
开启一个 OpenBSD 端口
开启一个 AUX 端口
开启一个 Cygnus 端口
添加字符串函数到回归套件 (Thomas)
扩展一些以前被截断为 16 个字符的函数名 (Thomas)
删除不需要的 malloc() 调用并用 palloc() 替换 (Bruce)
```

E.233. 版本 6.2.1

> 发布日期: 1997-10-17

6.2.1 是 6.2 的除错版本和增强可用性版本。

摘要：

- 允许字符串跨行，类似 SQL92。
- 包括了一个触发器的样例函数，用于在更新表时插入用户名。

这是对6.2的一个小的除错版本。对于从 6.2 以前的系统上升级， 需要进行一次完整的转储/恢复工作。请参考 6.2 版本信息获取相关指导。

E.233.1. 从 v6.2 迁移到 v6.2.1

这是一次小的除错升级。从v6.2上升级不需要进行转储/恢复，但对任何v6.2以前的版本是需要的。

当你从 v6.2 上升级时，如果你选择了转储/恢复的做法。你将会发现 avg(money) 现在可以正确运算了。所有其他的除错在升级了可执行程序后都将生效。

另一个避免转储/恢复的做法是在 `psql` 中使用下面的 SQL 命令来升级现有的系统表：

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

我们需要对包括 template1 在内的所有现有数据库进行上面操作。

E.233.2. 修改列表

```
允许 TIME 和 TYPE 字段名 (Thomas)
允许 true/false 的更大范围作为布尔值 (Thomas)
支持 "now" 和 "current" 的输出 (Thomas)
适当的处理带有 NULL 的 INSERT 的 DEFAULT (Vadim)
修复缓冲区管理中的关系参考计数的问题 (Vadim)
允许字符串跨行，如 ANSI (Thomas)
修复带有 ORDER BY 的向后的游标 (Vadim)
修复 avg(cash) 计算 (Thomas)
修复在 ORDER/GROUP BY 中指定一个字段两次 (Vadim)
记录新的 libpq 函数返回受影响的行，PQcmdTuples (Bruce)
INSERT/UPDATE 插入用户名的触发器函数 (Brook Milligan)
```

E.234. 版本 6.2

> 发布日期: 1997-10-02

希望从以前版本的 PostgreSQL 升级到此版本的用户需要对数据库做一转储/恢复的工作。

E.234.1. 从版本 6.1 迁移到版本 6.2

这种迁移需要对6.1的数据库进行一次完全的转储，然后再恢复到6.2的数据库里面。

请注意，6.2 的 `pg_dump` 和 `pg_dumpall` 工具应该用来转储 6.1 数据库。

E.234.2. 从版本 1. `_x_` 迁移到版本 6.2

任何从早于 1.* 版本的升级都要首先升级到 1.09， 因为在 1.02 版本中对 COPY 的输出格式进行了改进。

E.234.3. 修改列表

```
Bug 修复
---------
修复 pg_dump 继承、序列、归档表的问题 (Bruce)
修复由于 Solaris 中转换、无符号和坏的原型引起的溢出导致的编译错误 (Diab Jerius)
修复几何线算法的 bug (损坏的交叉计算) (Thomas)
检查在端点处的几何交叉以避免圆整 (Thomas)
捕捉非功能性的删除尝试 (Vadim)
更改时间函数名使其更一致 (Michael Reifenberg)
检查零分 (Michael Reifenberg)
检查一个命令对其本身可见的行更改/插入（这样我们有多个更新行的更新，等）的非常老旧的bug (Vadim)
修复 SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD 现在被允许了 (Patrick)
从 contrib/pginterface 中删除不需要的信号的东西
修复了 OR (where x != 1 or x isnull didn't return rows with x NULL) (Vadim)
修复了 time_cmp 函数 (Vadim)
修复了在 WHERE 子句中处理带有非属性第一参数的函数 (Vadim)
修复了记录的顺序和目标列表的顺序不同时的 GROUP BY (Vadim)
修复了不带有 sfunc1 的聚集的 pg_dump (Vadim)

增强
------------
默认遗传优化器 GEQO 参数现在是 8 个 (Bruce)
允许在函数中的有聚集的目标列表中使用参数 (Vadim)
添加 JDBC 驱动器作为一个接口 (Adrian & Peter)
pg_password 工具
返回 INSERT/UPDATE/DELETE 等 inserted/affected 的行数 (Vadim)
触发器由 CREATE TRIGGER 实现 (SQL3)(Vadim)
SPI (Server Programming Interface) 允许查询在 C 函数内部执行 (Vadim)
实现了 NOT NULL (SQL92)(Robson Paniago de Miranda)
为字符串处理、外连接和联合包括保留字 (Thomas)
使用唯一状态实现了扩展的注释 ("/* ... */") (Thomas)
添加 "//" 单行注释 (Bruce)
删除操作符名字里的一些字符上的限制 (Thomas)
实现了表的 DEFAULT 和 CONSTRAINT (SQL92)(Thomas)
```

```
添加文本串联操作符和函数 (SQL92)(Thomas)
支持 WITH TIME ZONE 语法 (SQL92)(Thomas)
支持 INTERVAL unit TO unit 语法 (SQL92)(Thomas)
定义类型 DOUBLE PRECISION, INTERVAL, CHARACTER, 和 CHARACTER VARYING (SQL92)(Thomas)
定义类型 FLOAT(p) 和基本的 DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
定义 EXTRACT(), POSITION(), SUBSTRING(), 和 TRIM() (SQL92)(Thomas)
定义 CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
为 UNION, HAVING, INNER 和 OUTER JOIN 添加语法和警告 (SQL92)(Thomas)
添加更多保留字，大部分是 SQL92 兼容的 (Thomas)
允许 timespan/reltime 类型的 hh:mm:ss 时间选项 (Thomas)
为 lseg, path, polygon 添加 center() 例程 (Thomas)
为 circle-polygon, polygon-polygon 添加 distance() 例程 (Thomas)
明确的检查使用轴线交叉算法的多边形包含的点和多边形 (Thomas)
添加转换 circle-box 的例程 (Thomas)
合并不同几何数据类型冲突的操作符 (Thomas)
用 "<->" 替换距离操作符 "<===>" (Thomas)
用">^" 替换 "above" 操作符 "!^" 和用 "<^" 替换 "below" 操作符 "!|" (Thomas)
为文本在两端截取、子字符串和字符串位置添加例程 (Thomas)
添加转换程序 circle(box) 和 poly(circle) (Thomas)
允许以内存而不是文件排序的内部排序 (Bruce & Vadim)
允许在内部同一类型上的函数和操作符成功 (Bruce)
加速剖像分析之后的后端启动 (Bruce)
为了性能内联频繁的调用函数 (Bruce)
减少 open() 调用 (Bruce)
psql:  添加 PAGER 为了 \h 和 \?,\C 修复
修复没有 tty 时的 psql pager (Bruce)
新增 entab 工具 (Bruce)
为参照完整性添加一般触发器函数 (Vadim)
为时间间隔添加一般触发器函数 (Vadim)
为 AUTOINCREMENT/IDENTITY 特性添加一般触发器函数 (Vadim)
MOVE 实现 (Vadim)

源代码树的变化
-------------------
HP-UX 10 补丁 (Vladimir Turin)
添加 SCO 支持 (Daniel Harris)
MkLinux 补丁 (Tatsuo Ishii)
更改几何盒子术语从 "length" 到 "width" (Thomas)
反对几何学代码中临时的未存储的斜率字段 (Thomas)
从 INSTALL 中删除重启指令 (Bruce)
安装时首先查看 /usr/ucb (Bruce)
修复 c++ 拷贝示例代码 (Thomas)
添加 -o 到 psql 手册页 (Bruce)
阻止未分配字符串长度的真实名被拷贝到数据库 (Bruce)
清理 NAMEDATALEN 的使用 (Bruce)
修复在输出中超过 15 个字符的 pg_proc 名 (Bruce)
添加 strNcpy() 函数 (Bruce)
删除一些不需要的 (void) 转换 (Bruce)
新增接口路径 (Marc)
用到 fd.c 函数的调用替换 fopen() 调用 (Bruce)
尽可能使函数静态 (Bruce)
将不适用的函数放在 #ifdef NOT_USED 中 (Bruce)
删除时间戳支持中对 difftime() 的调用以修复 SunOS (Bruce & Thomas)
修改 Digital Unix
pg_dumpall 的可移植性修复 (Bruce)
重命名 pg_attribute.attnvals 为 attdispersion (Bruce)
"intro/unix" 手册页现在是 "pgintro" (Bruce)
"built-in" 手册页现在是 "pgbuiltin"(Bruce)
"drop" 手册页现在是 "drop_table"(Bruce)
添加 "create_trigger", "drop_trigger" 手册页 (Thomas)
添加约束回归测试 (Vadim & Thomas)
添加注释语法回归测试 (Thomas)
添加 PGINDENT 并支持编程 (Bruce)
大量的提交在所有的 *.c 和 *.h 文件上运行 PGINDENT (Bruce)
文件移动到了 /src/tools 目录 (Bruce)
SPI 和触发器编程指南 (Vadim & D'Arcy)
```

E.235. 版本 6.1.1

> 发布日期: 1997-07-22

E.235.1. 从版本 6.1 迁移到版本 6.1.1

这是一个小的除错升级版本。从v6.1的版本到此版本的升级不需要进行转储/恢复的工作， 但是任何从早于 v6.1 的升级需要做这个工作。请参考v6.1的版本信息获取更多细节。

E.235.2. 修改列表

```
修复带有选项的 SET (Thomas)
允许 pg_dump/pg_dumpall 保留对所有表/对象的所有权 (Bruce)
新增 psql \connect 选项允许改变用户名而不改变数据库
修复 initdb --debug 选项(Yoshihiko Ichikawa)
lextest 清理 (Bruce)
hash 修复 (Vadim)
修复日期/时间月份边界算法 (Thomas)
为一些接口修复时区白天处理 (Thomas, Bruce, Tatsuo)
彻底检修了时间戳，以使用标准函数 (Thomas)
其他日期/时间例程中的代码清理 (Thomas)
psql 的 \d 现在大小写不敏感了 (Bruce)
psql 的反斜杠命令现有可以有尾随的分号 (Bruce)
修复使用 \g 时 psql 中的内存泄露 (Bruce)
主要修复尾数法处理到服务器的通信 (Thomas, Tatsuo)
修复 Solaris 汇编程序和包括文件 (Yoshihiko Ichikawa)
允许在用户名中有下划线 (Bruce)
pg_dumpall 现在返回适当的状态，可移植性修复 (Bruce)
```

E.236. 版本 6.1

> 发布日期: 1997-06-08

回归测试已经适应并且为 PostgreSQL 版本 6.1 做了大量的修改。

增加了三种新的 PostgreSQL 内部数据类型（`datetime`, `timespan`, 和 `circle`）。统一了 Points，boxes, paths,和 polygons 的输出格式。 在 misc.out 中的 polygon （多边形）的输出只是相对原先的回归测试输出进行了抽样检查。

PostgreSQL 6.1 提供了一个可选的使用基因算法的优化器。 这些算法在对包含多个限定或多个表（优化器需要对评估的排序选择时） 的查询输出进行排序时表现得更为随机。有好几个回归测试项目显式地修改了结果的排序， 这样就对优化器选择不敏感了。有几个对数据类型的回归测试本来就是乱序的（如点和时间间隔）， 包含这些类型的测试明显和 `set geqo to 'off'` 和 `reset geqo` 相等。

对数组说明符（包围在原子值周围的花括号）的解释看起来在回归测试产生之后的某个时候被改变了。 现在的 `./expected/*.out` 文件反映了这个新的解释，但却有可能是错误的！

float8 的回归测试至少在某些平台上会失败。这是因为对 `pow()` 和 `exp()` 的不同的实现方法以及用于溢出和下溢（underflow）条件的信号机制的不同造成的。

在随机测试中的"随机"结果回导致"随机"测试"失败"， 因为回归测试是简单的用 diff 进行比较的。不过，"随机"在我的机器上 (Linux/gcc/i686)看起来好象并没有产生随机结果。

E.236.1. 迁移到版本 6.1

迁移需要对数据库进行完整的6.0版本的转储和6.1版本的恢复。

对于早于1.*的版本：首先要升级到1.09版本，因为在1.02版本中已经改善了 COPY 的输出格式。

E.236.2. 修改列表

```
Bug 修复
---------
在库程序中检查包的长度
锁管理的优先级修补
检查 float8 的上/下溢 (Bruce)
多个表连接的修复 (Vadim)
SIGPIPE 崩溃修复 (Darren)
大对象修复 (Sven)
允许 btree 索引处理 NULL (Vadim)
时区修复 (D'Arcy)
select SUM(x) 可以在没有行时返回 NULL (Thomas)
内部优化，执行 bug 修复 (Vadim)
```

```
修复 < 或 <= 中的内循环没有行的问题 (Vadim)
阻止重新连接索引子句 (Vadim)
修复多个表的连接子句 (Vadim)
修复哈希，为数组 hashjoin (Vadim)
为 abstime 类型修复 btree (Vadim)
大对象修复 (Raymond)
修复哈希索引中的缓存泄露 (Vadim)
修复用于内部扫描的 rtree (Vadim)
修复用于内部扫描的 gist，清理 (Vadim, Andrea)
避免不必要的本地缓冲区分配 (Vadim, Massimo)
修复事务退出时本地缓冲区泄露 (Vadim)
修复文件管理内存泄露，清理 (Vadim, Massimo)
修复存储管理内存泄露 (Vadim)
修复 btree 重复处理 (Vadim)
修复由于 vacuum 导致已删除的行重现 (Vadim)
修复 SELECT varchar()/char() INTO TABLE  制造零长度的字段 (Bruce)
使用 Purify 修复许多 psql, pg_dump, 和 libpq 内存泄露 (Igor)

增强
------------
属性最优化统计 (Bruce)
更快的新 btree 批量加载代码 (Paul)
BTREE UNIQUE 添加到批量加载代码 (Vadim)
新增锁调试代码 (Massimo)
libpg++ 有大量的改变 (Leo)
新增 GEQO 优化器加速表多个表优化 (Martin)
新增 WARN 消息，为非唯一的数据插入到唯一键中 (Marc)
update x=-3，没有空格，现在有效了 (Bruce)
删除大小写敏感的标识符的处理 (Bruce,Thomas,Dan)
调试后端现在以树的形式打印 (Darren)
新增 Oracle 字符函数 (Edmund)
新增明文口令函数 (Dan)
no such class 或 insufficient privilege 更改为不同的消息 (Dan)
新增 ANSI 时间戳函数 (Dan)
新增 ANSI 时间和日期类型 (Thomas)
移动后端中的大的数据块 (Martin)
多字段 btree 索引 (Vadim)
新增 SET var TO value 命令 (Martin)
在需要时更新事务状态 (Dan)
为字符类型新增本地设置 (Oleg)
新增 SEQUENCE 序列号生成器 (Vadim)
GROUP BY 函数现在是可能的 (Vadim)
重组回归测试 (Thomas,Marc)
新增优化器操作加权 (Vadim)
新增 psql \z grant/permit 选项 (Marc)
新增 MONEY 数据类型 (D'Arcy,Thomas)
tcp 套接字通信速度提升 (Vadim)
为属性状态和特定字段新增 VACUUM 选项 (Vadim)
改善了许多几何类型 (Thomas,Keith)
附加了回归测试 (Thomas)
新增 datestyle 变量 (Thomas,Vadim,Martin)
为排序类型添加了更多比较操作符 (Thomas)
新增转换函数 (Thomas)
新增更多简洁的 btree 格式 (Vadim)
允许 pg_dumpall 保存数据库所有权 (Bruce)
新增 SET GEQO=# 和 R_PLANS 变量 (Vadim)
旧的 (!GEQO) 优化器可以使用右侧规划 (Vadim)
改善了 SQL 分析器中的类型检查 (Bruce)
新增 SET, SHOW, RESET 命令 (Thomas,Vadim)
新增 \connect database USER 选项
新增 destroydb -i 选项 (Igor)
新增 \dt 和 \di psql 命令 (Darren)
SELECT "\n" 现在逃逸新行 (A. Duursma)
新增老的格式的几何转换函数 (Thomas)

源代码树的改变
-------------------
新增配置脚本 (Marc)
添加了 readline  配置选项 (Marc)
删除了 OS 特定的配置选项 (Marc)
新增 OS 特定的模板文件 (Marc)
不再需要编辑 Makefile.global (Marc)
```

```
重新安排包含文件 (Marc)
nextstep 补丁 (Gregor Hoffleit)
删除了 Windows 特定的代码 (Bruce)
删除了 postmaster -e 选项, 现在只有 postgres -e 选项 (Bruce)
合并 front/backends 中重复的库代码 (Martin)
现在使用 eBones, international Kerberos(Jun)
支持更多的共享库
清理 c++ 包含文件 (Bruce)
警告 buggy flex (Bruce)
DG/UX, Ultrix, IRIX, AIX 可移植性修复
```

E.237. 版本 6.0

> 发布日期: 1997-01-29

对于想从此版本以前的 PostgreSQL 迁移到此版本的都需要做转储/恢复工作。

E.237.1. 从版本 1.09 迁移到版本 6.0

这种迁移需要对1.09数据库完全转储，然后在6.0里将数据全部恢复回去。

E.237.2. 从 1.09 以前的版本迁移到版本 6.0

从 1.* 以前的版本迁移首先要升级到 1.09 版本，因为在 1.02 版本中已经改善了 COPY 的输出格式。

E.237.3. 变化

```
Bug 修复
---------
ALTER TABLE bug - 运行 postgres 进程需要重读表定义
允许 vacuum 运行在一个表或整个数据库上 (Bruce)
修复数组
修复数组内存写入溢出 (Kurt)
修复难懂的 btree range/non-range bug (Dan)
修复在某些类型如时间和日期上的哈希索引
修复 pg_log 尺寸激增
修复 lo_export() 上的权限 (Bruce)
修复未初始化的读取内存 (Kurt)
修复 ALTER TABLE ... char(3) bug (Bruce)
修复一些小的内存泄露
修复 EXPLAIN 处理选项和改变了的 full_path 选项名
修复 acl 权限分组的输出
内存泄露（像 Purify 这样的工具的破坏） (Kurt)
规则系统小幅度的改善
修复 NOTIFY
为运行时检查新增 asserts
彻底检查 parser/analyze 代码以适当的报告错误和提升速度
Pg_dump -d 现在可以适当的处理 NULL (Bruce)
阻止 SELECT NULL 使服务器崩溃 (Bruce)
当 INSERT ... SELECT columns 不匹配时适当的报告错误
当插入字段名不正确时适当的报告错误
psql \g filename 现在可以工作了 (Bruce)
psql 修复一行上的多个声明有多个输出时的问题
删除了重复的系统 OID
SELECT * INTO TABLE . GROUP/ORDER BY 如果表已经存在则给出未连接的错误 (Bruce)
修复了几个使后端崩溃的查询
插入字符串中起始引号错误 (Bruce)
提交空查询现在返回空状态，不只是 " " 查询 (Bruce)

增强
------------
添加 EXPLAIN 手册页 (Bruce)
添加 UNIQUE 索引功能 (Dan)
```

```
添加 hostname/user 级别访问控制而不只是 hostname 和 user
为 <> 添加同义词 != (Bruce)
允许 "select oid,* from table"
允许 BY,ORDER BY 通过编号指定字段，或通过非别名的 table.column (Bruce)
允许从前端 COPY (Bryan)
允许 GROUP BY 使用别名字段名 (Bruce)
允许真实的压缩，不只是在同一页上重新使用 (Vadim)
允许安装配置选项自动添加到本地用户 (Bryan)
允许 libpq 区分文本值 '' 和 null (Bruce)
允许非 postgres 用户在 destroydb 上有 createdb 权限
允许限制谁可以创建 C 函数 (Bryan)
允许限制谁可以在后端 COPY (Bryan)
可以收缩表、pg_time 和 pg_log (Vadim & Erich)
改变调试级别 2 为只输出查询，改变调试标题布局 (Bruce)
改变缺省小数常量表示从 float4 到 float8 (Bruce)
当 postmaster 启动时设置 European 日期格式
如果没有找到准确的大小写情况则执行小写的函数名
修复 aggregate/GROUP 处理，允许 'select sum(func(x),sum(x+y) from z'
Gist 现在包含在发布里 (Marc)
本地用户的身份认证 (Bryan)
实现了 BETWEEN 限定符 (Bruce)
实现了 IN 限定符 (Bruce)
libpq 有 PQgetisnull() (Bruce)
改善了 libpq++
新增选项到 initdb (Bryan)
Pg_dump 允许转储 OID (Bruce)
Pg_dump 为了速度在加载表之后创建索引 (Bruce)
Pg_dumpall 转储所有数据库，和用户表
为 NULL 值附加 Pginterface (Bruce)
阻止 postmaster 作为 root 运行
psql \h 和 \? 现在可读 (Bruce)
psql 允许在行中任何地方有反向斜线，分号 (Bruce)
psql 改变了查询中或引号中行的命令提示符 (Bruce)
psql char(3) 现在在 \d 输出中作为 (bp)char 显示 (Bruce)
psql 现在返回更精确的代码 (Bryan?)
psql 更新帮助语法 (Bruce)
重复访问和修复 vacuum (Vadim)
减少回归差异的大小，删除时区名差异 (Bruce)
删除编译时参数以启用二进制发布 (Bryan)
HBA 标记的相反含义 (Bryan)
本地用户的安全认证 (Bryan)
加速 vacuum (Vadim)
vacuum 现在有了 VERBOSE 选项 (Bruce)

源代码树的改变
-------------------
所有函数现在有了和调用相比较的原型
允许维护可以简单的在 Makefile.global 中禁用 (Bruce)
改变在代码中使用的 oid 常量为 #define 名
解耦 sparc 和 solaris 的定义 (Kurt)
Gcc -Wall 编译只有来自不固定结构的警告
主要包括文件 reorganization/reduction (Marc)
Make 现在在编译失败时停止 (Bryan)
Makefile 重组 (Bryan, Marc)
合并 bsdi_2_1 到 bsdi (Bruce)
删除了监控程序
更改名字 Postgres95 为 PostgreSQL
新增 config.h 文件 (Marc, Bryan)
PG_VERSION 现在设置为 6.0 并且被 postmaster 使用
可移植性附加物，包括 Ultrix, DG/UX, AIX, 和 Solaris
减少 #define 的数量，集中 #define
删除系统表中重复的 OID (Dan)
删除重复的系统目录信息或报告不匹配 (Dan)
删除许多 os 特定的 #define
重组目标文件 generation/location (Bryan, Marc)
重组端口特定的文件位置 (Bryan, Marc)
未使用的/未初始化的变量纠正
```

E.238. 版本 1.09

> 发布日期: 1996-11-04

抱歉，我们没有追踪从 1.02 到 1.09 的变更。 有些在 6.0 里列出来的修改实际上包含在 1.02.1 到 1.09 的版本中。

E.239. 版本 1.02

> 发布日期: 1996-08-01

E.239.1. 从版本 1.02 迁移到版本 1.02.1

本文是用于 1.02.1 的新的迁移文件。它包括 'copy' 的改变和一个用于转换旧的 ASCII 文件的脚本。

> **Note:** 下面的信息用于帮助那些希望从 Postgres95 1.01 和 1.02 向 Postgres95 1.02.1 迁移的用户。
>
> 如果你刚刚开始使用 Postgres95 1.02.1 并且不需要迁移旧的数据库， 那么你不需要阅读下面的部分。

要想从旧的 Postgres95 版本 1.01 或 1.02 数据库向版本 1.02.1 升级， 需要进行下面步骤：

1. 运行新的 1.02.1 postmaster
2. 向 1.01 或 1.02 数据库中增加 1.02.1 的新的内建函数和操作符。 方法是在你的 1.01 或 1.02 的数据库上运行新的 1.02.1 服务器然后应用文件末尾的查询。 这些工作可以很轻松的通过 `psql` 来完成。如果你的 1.01 或 1.02 数据库叫 `testdb` 并且你已经把文件末尾的命令剪切下来并存入到文件 `addfunc.sql` 里，那么：

   ```
   % psql testdb -f addfunc.sql
   ```

   那些从 1.02 数据库升级的人们在执行文件中最后两个语句时会看到警告信息， 因为它们已经在 1.02 中出现了。 这些警告不会产生问题。

E.239.2. 转储/重装过程

如果你试图重装 pg_dump 或文本模式， `copy tablename to stdout` 会在原先的版本中生成，你将需要在加载到数据库之前在 ASCII 文件上运行下面的 `sed` 脚本。 因为原先的格式用'.'作为数据结束符，而现在使用'.'。同样，现在空字符串用"进行装载， 而不是 NULL。请参阅 copy 的手册页获取完整的细节描述。

```
sed 's/^\.$/\\./g' <in_file >out_file
```

如果你从一个旧的二进制拷贝或非标准输出拷贝中恢复数据， 就不需要做上述转换，因为不存在数据结束符字符问题。

```
-- agc添加的下列的行反应不区分大小写
-- 正则表达式搜索 varchar (in 1.02), 和 bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

E.239.3. 修改列表

```
源代码管理和开发
 * 世界各地的志愿者团队
 * 源代码树现在在 CVS 在 ftp.ki.net

增强
 * psql (和底层的 libpq 库)现在对格式化输出有了更多的选项, 包括 HTML
 * pg_dump 现在输出模式和/或数据, 带有许多提高完善的修复。
 * psql 在管理 shell 脚本中用来替代监控器。监控器将在下一个版本中停用。
 * 日期/时间函数增强
 * NULL 插入/更新/比较 修复/增强
 * TCL/TK 库和 shell 修复以和 tck7.4/tk4.0 和 tcl7.5/tk4.1 一起工作

Bug 修补 (多得说不完)
 * 索引
 * 存储管理
 * 在解除参考之前检查 NULL 指针
 * Makefile 修复

新增端口
 * 添加了 SolarisX86 端口
 * 添加了 BSD/OS 2.1 端口
 * 添加了 DG/UX 端口
```

E.240. 版本 1.01

> 发布日期: 1996-02-23

E.240.1. 从版本 1.0 迁移到版本 1.01

下面信息是给那些希望将数据库从Postgres95 1.0 向 Postgres95 1.01 迁移的用户的一些有用信息。

如果你是刚刚安装完成 Postgres95 1.01 并且没有需要迁移的旧数据库， 那么你不需要阅读下面部分。

如果要把 Postgres95 版本 1.0 的数据库向 Postgres95 版本 1.01 迁移，需要进行下面的步骤：

1. 把文件 `src/Makefile.global` 里的变量 `NAMEDATALEN` 定义为16， `OIDNAMELEN` 定义为20。
2. 决定自己是否需要以主机为基础的认证（HBA）。
   i. 如果你需要这么做，你必须在顶级数据目录（通常是你的环境变量 `$PGDATA` 的值）里创建一个名为 `pg_hba` 的文件。 我们在例子语法里用 `src/libpq/pg_hba` 代表。
   ii. 如果你不需要这样以主机为基础的认证，你可以把 `src/Makefile.global` 里的下面这行注释掉

   ```
   HBA = 1
   ```

   要注意缺省时以主机为基础的认证（HBA）是打开的， 而且如果你不做上面所说的步骤A或B中的其中一步，其他主机上（out-of-the-box） 的1.01版本将不允许你与1.0的数据库联接。
3. 编译和安装 1.01，但是不要执行 `initdb` 步骤。
4. 在进行下一步之前，终止 1.0 的 postmaster 进程，然后备份你现有的 `$PGDATA` 目录。
5. 把你的 `PGDATA` 环境变量设置为你的 1.0 的库（的位置）， 但是把路径设置成 1.01 的可执行文件路径。
6. 把文件 `$PGDATA`/PG_VERSION 从 5.0 修改成 5.1
7. 运行新的 1.01 的 postmaster。

8. 把 1.01 的新的内建的函数和操作符追加到 1.0 的数据库中去。 这一步是通过在你的 1.0 的库上运行 1.01 的服务器，并且运行附加的查询并保存到文件 1.0_to_1.01.sql 中。 如果你的 1.0 数据库名为 `testdb`，那么我们可以通过 `psql` 很容易完整升级工作：

   ```
   % psql testdb -f 1.0_to_1.01.sql
   ```

   然后执行下面命令（可以从下面剪切和拷贝）：

   ```
   -- add builtin functions that are new to 1.01

   create function int4eqoid (int4, oid) returns bool as 'foo'
   language 'internal';
   create function oideqint4 (oid, int4) returns bool as 'foo'
   language 'internal';
   create function char2icregexeq (char2, text) returns bool as 'foo'
   language 'internal';
   create function char2icregexne (char2, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexeq (char4, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexne (char4, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexeq (char8, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexne (char8, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexeq (char16, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexne (char16, text) returns bool as 'foo'
   language 'internal';
   create function texticregexeq (text, text) returns bool as 'foo'
   language 'internal';
   create function texticregexne (text, text) returns bool as 'foo'
   language 'internal';

   -- add builtin functions that are new to 1.01

   create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
   create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
   create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
   create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
   create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
   create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
   create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
   create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
   create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
   create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
   create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
   create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);
   ```

E.240.2. 修改列表

```
不兼容性：
 * 1.01 向后兼容 1.0 数据库，提供了用户指导步骤，在 MIGRATION_from_1.0_to_1.01 文件中概述。
   如果没有采取这些步骤，1.01 是不与 1.0 数据库兼容的。

增强：
 * 添加了 PQdisplayTuples() 到 libpq 并且为了使用它更改了监控器和 psql
 * 添加了 NeXT 端口 (需要 SysVIPC 实现)
 * 添加了 CAST .. AS ... 语法
 * 添加了 ASC 和 DESC 关键字
 * 添加了 'internal' 作为 CREATE FUNCTION 内部函数的可能语言，CREATE FUNCTION 内部函数是 C 函数，
   已经静态的连接到了 Postgres 后端。
 * 为系统标识符添加了一个新的类型 "name" （表名，属性名等）。这个替换老的 char16 类型。
   通过在 src/Makefile.global 中的 NAMEDATALEN #define 设置。
 * 一个可读的参考手册描述查询语言。
 * 添加了基于主机的访问控制。一个配置文件($PGDATA/pg_hba)用来保存配置数据。
   如果基于主机的访问控制不再需要了，注释掉 src/Makefile.global 中的 HBA=1。
 * 更改正则表达式处理为统一的使用 Henry Spencer 的正则表达式代码，不管是什么平台。
   正则表达式代码包含在发布中。
 * 为大小写不敏感的正则表达式添加了函数和操作符。操作符是 ~* 和 !~*。
 * pg_dump 为了更好的性能使用 COPY 而不是 SELECT 循环

Bug 修复：
 * 修复了一个优化器 bug ，当函数调用在 WHERE 子句中用于比较时会引起内核转储
 * 更改所有 getuid 的使用为 geteuid ，这样就使用了有效的 uid
 * psql 在使用 -C 发生错误时返回非零的状态
 * 应用了公共补丁 1-14
```

E.241. 版本 1.0

> 发布日期: 1995-09-05

E.241.1. 修改列表

```
版权修改：
 * Postgres 1.0 的版权已经修改放宽到可以自由修改和可以为任何目的自由修改。
   请参阅 COPYRIGHT 文件。感谢 Michael Stonebraker 教授令这些成为可能。

不兼容性：
 * 数据格式必须是 MM-DD-YYYY (或如果你使用 EUROPEAN STYLE 则为 DD-MM-YYYY)。遵从 SQL-92 规格。
 * "delimiters" 现在是一个关键字

增强：
 * 添加了 sql LIKE 语法
 * copy 命令现在接受一个选项 USING DELIMITER 。分隔符可以是任意的单字符字符串。
 * 添加了 IRIX 5.3 端口。感谢 Paul Walmsley 和其他人。
 * 升级了 pg_dump，使其可以与新的 libpq 一起工作
 * \d 已经加入了 psql 中，感谢 Keith Parks
 * 由于预编译模式的缓存，使用 POSIX 正则表达式的架构的正则表达式性能已经改善了。感谢 Alistair Crooks
 * libpq++ 的一个新版本。感谢 William Wanders

Bug 修复：
 * 任意的 userids 可以在 createuser 脚本中指定
 * \c 连接到其他现在正在 psql 中工作的数据库。
 * 修复了 float4inc() 的坏的 pg_proc 记录
 * 带有 usecreatedb 字段设置的用户现在可以创建数据库，不必是超级用户
 * 当记录不再有任何特权时删除访问控制记录
 * 修复了不方便的日期时间实现
 * 添加了 kerberos 标识到 src/backend/Makefile
 * libpq 现在与 kerberos 一起工作
 * 用户手册上的排版错误已经纠正了
 * 带有多个索引的 btree 从不工作，现在我们告诉你当你尝试使用它们时它们不会工作
```

E.242. Postgres95 版本 0.03

> 发布日期: 1995-07-21

E.242.1. 修改列表

```
不兼容的变化：
 * BETA-0.3 与之前版本创建的数据库不兼容（由于系统目录和索引结构的改变）
 * 双引号(")作为字符串文本的引用字符已经弃用了；你需要将它们转换为单引号(')。
 * 聚集名（如 int4sum）为了与 SQL 标准（如 sum）一致已经重命名了
 * CHANGE ACL 语法被 GRANT/REVOKE 语法替代了
 * 浮点值（如 3.14）现在是 float4 类型（而不是在以前版本中的 float8）；
   如果你确定是 float8，你可能必须类型转换。如果你忘记了做类型转换，
   并且你分配一个 float 文本到一个 float8 类型的字段，你可能会得到不正确的存储值！
 * LIBPQ 已经完全修补了，这样前端应用可以连接到多个后端
 * pg_user 中的 usesysid 字段已经从 int2 改变成 int4，以允许 Unix 用户 id 有更广泛的范围。
 * netbsd/freebsd/bsd o/s 端口已经合并到了一个单一的 BSD44_derived 端口。（感谢 Alistair Crooks

 SQL 标准兼容(下列详细的改变令 postgres95 与 SQL-92 标准更加兼容):
   * 下列的 SQL 类型现在是内建的：smallint, int(eger), float, real,
   char(N), varchar(N), date 和 time。

   下列是现存 postgres 类型的别名：
                smallint -> int2
                integer, int -> int4
                float, real  -> float4

   char(N) 和 varchar(N) 作为截断的文本类型实施。另外，char(N) 有空白填充。

 * 单引号(')用于引用字符串字面值；'' （除了 \'）作为在一个字符串中插入一个单引号的含义来支持
 * 使用 SQL 标准聚集名(MAX, MIN, AVG, SUM, COUNT)（还有，聚集现在可以重载，也就是，
   你可以定义你自己的 MAX 聚集来接受一个用户定义的类型。）
 * 删除了 CHANGE ACL。添加了 GRANT/REVOKE 语法。
   - 权限可以通过使用 "GROUP" 关键字赋予一个组。

例如：
                GRANT SELECT ON foobar TO GROUP my_group;

关键字 'PUBLIC' 也支持意为所有用户。

权限一次只能赋予或撤销一个用户或组。

不支持 "WITH GRANT OPTION"。只有类的所有者可以改变访问控制。

   - 缺省的访问控制赋予用户只读访问。你必须明确授予用户 insert/update 访问。
     要改变这个，在定义 ACL_WORLD_DEFAULT 中更改行：
                src/backend/utils/acl.h

Bug 修复：
 * 聚集在空表处不运行的bug已经修复了。现在，聚集在空表上运行时会返回聚集的初始状态。
   因此，对一个空表 COUNT 现在可能会返回 0。对一个空表  MAX/MIN 将会返回一个值为 NULL 的行。
 * 允许在监控器中使用 \;
 * LISTEN/NOTIFY 异步通知机制现在可以工作了
 * 规则动作体中的 NOTIFY 现在可以工作了
 * 哈希索引现在可以工作了，访问方法一般来说应该执行的更好了。大的 btree 索引的创建应该更快了。
   （感谢 Paul Aoki）

其它修改和增强：
 * 添加了用于解释查询执行规划的 EXPLAIN 语句（如 "EXPLAIN SELECT * FROM EMP"输出该查询的执行规划）。
 * WARN 和 NOTICE 消息不在带有时间戳。要启用错误消息的时间戳，取消 src/backend/utils/elog.h 中下列
        /* define ELOG_TIMESTAMPS */
```

```
 * 在违反访问控制时，会给出 "Either no such class or insufficient privilege" 消息。
   当没有找到类时给出同样的消息。这让没有权限的用户猜测有权限的用户的存在。
 * 一些用户不可见的附加系统目录已经做出了改变。

libpgtcl 修改：
 * -oid 选项已经添加到了 "pg_result" tcl 命令中。pg_result -oid 返回最后插入的行的 oid。
   如果最后的命令不是 INSERT，那么 pg_result -oid 返回 ""。
 * 大对象接口可以作为 pg_lo* tcl 命令使用：pg_lo_open, pg_lo_close, pg_lo_creat, 等。

可移植性增强和新的端口：
 * flex/lex 问题已经清理了。现在，应该可以在任意平台上使用 flex 代替 lex。
   不用再假设基于你使用的平台使用什么 lexer。
 * 现在支持 Linux-ELF 端口了。已经检验了各种配置：下列的配置已知是通过了的：
    kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
   任何东西都是 ELF 格式

新工具：
 * ipcclean 添加到发布
   ipcclean 通常不需要运行，但是如果你的后端崩溃了并且导致共享内存段原地停留，ipcclean将为你清理它们。

新文档：
 * 保留了用户手册并添加了 libpq 文档。
```

E.243. Postgres95 版本 0.02

> 发布日期: 1995-05-25

E.243.1. 修改列表

```
不兼容的更改：
* 创建一个数据库的 SQL 语句是 'CREATE DATABASE' 而不是 'CREATEDB'。相似的，
  删除一个数据库的是 'DROP DATABASE' 而不是 'DESTROYDB'。不过，
  可执行文件 'createdb' 和 'destroydb' 的名字保持不变

新工具：
* pgperl - a Perl (4.036) 接口到 Postgres95
* pg_dump - a 工具转储一个 postgres 数据库到一个包含查询命令的脚本文件。
  该脚本文件是 ASCII 格式的并且可以用来重建该数据库，即使是在其他机器和其他架构上。
  （转换 Postgres 4.2 数据库到 Postgres95 数据库也是很好的。）

下列接口已经并入 postgres95-beta-0.02 中了：
* Alistair Crooks 做的 NetBSD 接口
* Mike Tung 做的 AIX 接口
* Jon Forrest 做的 Windows NT 接口（更多东西但是还未完成）
* Brian Gallew 做的 Linux ELF 接口

下列bug在 postgres95-beta-0.02 中已经被修复了：
* 新行在 COPY OUT 中不逃逸和第一个属性是一个 '.' 时的 COPY OUT 的问题
* 在 createuser 中不能返回使用缺省的用户 id
* SELECT DISTINCT 在大表上时崩溃
* Linux 安装问题
* 监控器不允许 'localhost' 作为 PGHOST 使用
* 当进行 \c 或 \l 时 psql 内核转储
* "pgtclsh" 目标从 src/bin/pgtclsh/Makefile 中丢失
* libpgtcl 有一个硬链接的缺省端口号
* SELECT DISTINCT INTO TABLE 挂起
* CREATE TYPE 不接受 'variable' 作为 internallength
* 在一个 SELECT 中错误的结果使用多于 1 的聚集
```

E.244. Postgres95 版本 0.01

> 发布日期: 1995-05-01

最初发布版本。

Appendix F. 额外提供的模块

Table of Contents

本附录和下一个包含可以在PostgreSQL 发布的 `contrib` 目录中找到的模块信息。 这些包括移植工具，分析工具，和不是核心PostgreSQL系统部分的插件功能， 主要因为他们关注于有限的读者或者太实验而不是主要源码树的一部分。 这并不妨碍他们有效性。

该附录包含扩展以及在 `contrib` 中发现的服务器插件模块。 Appendix G包含实用程序。

当从源码发布中编译时， 这些组件不能自动创建， 除非你建立"world"目标(参阅step 2)。 你可以通过运行下面命令创建和安装：

```
<kbd class="literal">gmake</kbd>
<kbd class="literal">gmake install</kbd>
```

在配置的源码树的 `contrib` 目录中 或编译和安装一个选定的模块，在该模块的子目录做同样的事。 许多模块可以进行回归测试，可以通过运行下面命令被执行：

```
<kbd class="literal">gmake check</kbd>
```

在安装前或者

```
<kbd class="literal">gmake installcheck</kbd>
```

一旦你有正运行的PostgreSQL服务器。

如果您使用的是PostgreSQL的预包装版本， 这些模块通常可以作为一个单独的包，如 `postgresql-contrib` 。

许多模块提供新的用户自定义函数，操作符，或类型。 为了利用其中一个模块，在你安装代码之后 你需要在数据库系统中注册新的SQL对象。 在PostgreSQL 9.1和以后版本，这是通过执行 CREATE EXTENSION命令来实现。在一个新的数据库中， 你可以简单地做

```
CREATE EXTENSION _module_name_;
```

此命令必须由数据库管理员运行。这个 在当前数据库中注册新的SQL对象， 所以你需要在你想要该模 块的功能可用的每个数据库中运行这个 命令。另外， 在数据库 `template1` 中运行它，这样扩展将被复制到 缺省随后创建的数据库中。

许多模块允许你在你选择的模式中安装对象。 要做到这一点，添加 `SCHEMA _schema_name_` 到 `CREATE EXTENSION` 命令， 默认情况下，该对象将被放置在当前创建的 目标模式中，通常 `public` 。

如果你的数据库通过转储提出，并且从PostgreSQL9.1之前版本恢复， 你在它的模块的先前9.1版本使用过，你应该替代做

```
CREATE EXTENSION _module_name_ FROM unpackaged;
```

这将更新模块的9.1之前对象到一个适当的 扩展对象。 通过ALTER EXTENSION管理未来更新模块。 关于扩展更新的更多信息，参见Section 35.15。

注意，然而，这些模块在这个意义上不是"扩展"， 但在一些其他的方式中被加载到服务器，比如 通过 shared_preload_libraries。 参见每个模块的文档获取更多信息。

F.1. adminpack

`adminpack` 提供了一些pgAdmin与其他管理的支持函数， 并且管理工具可以 用于提供额外的功能，比如远程管理服务器的日志文件。

F.1.1. 函数实现

通过 `adminpack` 实现的该函数只能由超级用户运行。这里列出了这些函数：

```
int8 pg_catalog.pg_file_write(fname text, data text, append bool)
bool pg_catalog.pg_file_rename(oldname text, newname text, archivename text)
bool pg_catalog.pg_file_rename(oldname text, newname text)
bool pg_catalog.pg_file_unlink(fname text)
setof record pg_catalog.pg_logdir_ls()
/* 为了pgAdmin兼容性重命名已经存在的后端函数*/
int8 pg_catalog.pg_file_read(fname text, data text, append bool)
bigint pg_catalog.pg_file_length(text)
int4 pg_catalog.pg_logfile_rotate()
```

F.2. auth_delay

`auth_delay` 导致服务器在报告认证失败之前短暂停止， 使得暴力攻击数据库密码更难。 注意它并不能防止 拒绝服务攻击，甚至可能恶化它们， 因为报告验证失败之前等待的过程将损耗连接槽位。

为了该函数，这个模块必须通过 `postgresql.conf` 中的shared_preload_libraries 被加载。

F.2.1. 配置参数

`auth_delay.milliseconds` (`int`)

报告认证失败之前等待的毫秒数，缺省是0。

这些参数必须在 `postgresql.conf` 中设置。 典型的用法可能是：

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.2.2. 作者

KaiGai Kohei `<[kaigai@ak.jp.nec.com](mailto:kaigai@ak.jp.nec.com)>`

F.3. auto_explain

`auto_explain` 模块自动提供一种慢语句记录执行计划的方法 没有手动运行EXPLAIN。 这对于在大型应用程序中跟踪未优化查询尤其有用。

该模块没有提供SQL访问函数。为了使用它，简单加载它到服务器中。 你可以加载它到独立会话中：

```
LOAD 'auto_explain';
```

（你一定是超级用户才能这样做。） 更典型的用法是预先加载它到 在 `postgresql.conf` 中 shared_preload_libraries 包含 `auto_explain` 的所有会话。然后， 你可以出乎意料地跟踪无论什么时候会发生的慢查询。 当然有价值开销。

F.3.1. 配置参数

有几个配置参数控制 `auto_explain` 的操作。注意， 缺省操作是什么也不做，如果你想要任何结果， 那么你必须至少设置 `auto_explain.log_min_duration` 。

`auto_explain.log_min_duration` (`integer`)

`auto_explain.log_min_duration` 是最小声明执行时间， 以毫秒为单位，将导致语句的计划被记录。设置它为零记录所有计划。 减一（缺省）禁止计划的记录。比如，如果你设置它为 `250ms` ， 那么运行250ms或者更长时间的所有语句将被记录。只有超级用户可以改变这个设置。

`auto_explain.log_analyze` (`boolean`)

`auto_explain.log_analyze` 导致 `EXPLAIN ANALYZE` 输出，而不是 `EXPLAIN` 输出，当记录一个执行计划时，被打印。 该参数缺省是关闭的。只有超级用户可以改变这个设置。

> **Note:** 当该参数打开时，为所有执行语句产生每个计划节点时间， 无论他们是否运行足够长的时间来获得记录。 这会对性能产生极其不利的影响。

`auto_explain.log_verbose` (`boolean`)

`auto_explain.log_verbose` 导致 `EXPLAIN VERBOSE` 输出，而不是 `EXPLAIN` 输出，当记录一个执行计划时，被打印。 该参数缺省是关闭的。只有超级用户可以改变这个设置。

`auto_explain.log_buffers` (`boolean`)

`auto_explain.log_buffers` 导致 `EXPLAIN(ANALYZE, BUFFERS)` 输出，而不是 `EXPLAIN` 输出，当记录一个执行计划时，被打印。 该参数缺省是关闭的。只有超级用户可以改变这个设置。 该参数不起作用除非设置 `auto_explain.log_analyze` 参数。

`auto_explain.log_format` (`enum`)

`auto_explain.log_format` 选择使用 `EXPLAIN` 输出格式。 允许值是 `text` , `xml` , `json` 和 `yaml` 。 缺省是text。只有超级用户可以改变这个设置。

`auto_explain.log_timing` (`boolean`)

`auto_explain.log_timing` 导致 `EXPLAIN (ANALYZE, TIMING off)` 输出, 而不是 `EXPLAIN (ANALYZE)` 输出。 反复读取系统时钟的开销可以显著减缓一些系统上的查询，因此当计算实际行时，设置该 参数为关闭是非常有用的，不需要确切时间。 当启动 `auto_explain.log_analyze` 时，该参数是有效的。 缺省该参数是打开的。只有超级用户可以改变该设置。

`auto_explain.log_nested_statements` (`boolean`)

`auto_explain.log_nested_statements` 为了记录导致嵌套语句（语句在函数内执行）。 当它是关闭时，只记录顶级查询计划。 该参数缺省是关闭的。 只有超级用户可以改变这个设置。

必须在 `postgresql.conf` 中设置该参数，典型的用户是：

```
# postgresql.conf
shared_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.3.2. 例子

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

这可能产生日志输出比如：

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92 l
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual time=0.016
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92 l
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual time=
                      Filter: indisunique
```

F.3.3. 作者

Takahiro Itagaki
`<[itagaki.takahiro@oss.ntt.co.jp](mailto:itagaki.takahiro@oss.ntt.co.jp)>`

F.4. btree_gin

`btree_gin` 提供简单的GIN操作符类实现数据类型 `int2` , `int4` , `int8` , `float4` , `float8` , `timestamp with time zone` , `timestamp without time zone` , `time with time zone` , `time without time zone` , `date` , `interval` , `oid` , `money` , `"char"` , `varchar` , `text` , `bytea` , `bit` , `varbit` , `macaddr` , `inet` 和 `cidr` 的 B-tree等价操作。

总的来说，这些操作符类不会超过等值标准的B树索引方法， 他们缺乏标准B树代码的一个主要特点：强制唯一性的能力。然而， 它们对GIN测试是有用的，并且作为开发其他GIN操作符类的基础。 同时，对于查询，同时测试了GIN可索引列和B树可索引列， 对于创建使用这些操作符类之一的多列GIN索引比创建 通过位图与进行联合的两个独立索引更加有效。

F.4.1. 例子用法

```
CREATE TABLE test (a int4);
-- 创建索引
CREATE INDEX testidx ON test USING gin (a);
-- 查询
SELECT * FROM test WHERE a < 10;
```

F.4.2. 作者

Teodor Sigaev (<teodor@stack.net>)和 Oleg Bartunov (<oleg@sai.msu.su>)。参阅 http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin 获取额外信息。

F.5. btree_gist

`btree_gist` 提供 GiST索引操作符类实现数据类型 `int2` , `int4` , `int8` , `float4` , `float8` , `numeric` , `timestamp with time zone` , `timestamp without time zone` , `time with time zone` , `time without time zone` , `date` , `interval` , `oid` , `money` , `char` , `varchar` , `text` , `bytea` , `bit` , `varbit` , `macaddr` , `inet` 和 `cidr` B-tree等价操作。

总的来说，这些操作符类不会超过等值标准的B树索引方法，他们缺乏标准B树代码的一个主要特点：强制唯一性的能力。然而，它们提供一些不可用于B树索引的其他功能，正如下面描述的。另外当需要多列GiST索引时，这些操作符类是有用的，其中一些列是唯一可以使用GiST可索引的数据类型 但是其他列仅仅是简单的数据类型。最终，这些操作符类对GiST测试是有用的，并且作为开发其他GiST操作符类的基础。

除了典型的B树搜索操作符，`btree_gist` 还提供索引支持 `<>` ("不等")。这可能在与exclusion constraint结合中很用，正如下面描述。

另外，对于数据类型是一种天然的距离度量，`btree_gist` 定义了一个距离操作符 `<->` ，并且使用该操作符为最近相邻搜索提供GiST索引支持。为 `int2` , `int4` , `int8` , `float4` , `float8` , `timestamp with time zone` , `timestamp without time zone` , `time without time zone` , `date` , `interval` , `oid` 和 `money` 提供距离操作符。

F.5.1. 例子用法

使用 `btree_gist` 而不是 `btree` 的简单例子：

```
CREATE TABLE test (a int4);
-- 创建索引
CREATE INDEX testidx ON test USING gist (a);
-- 查询
SELECT * FROM test WHERE a < 10;
-- 最近相邻搜索：找到最接近"42"的十项
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

使用排斥约束可以执行该规则 动物园笼子里只能包含一种动物:

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING gist (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage, animal)=(123,
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.5.2. 作者

Teodor Sigaev (`<[teodor@stack.net](mailto:teodor@stack.net)>`), Oleg Bartunov (`<[oleg@sai.msu.su](mailto:oleg@sai.msu.su)>`)和 Janko Richter (`<[jankorichter@yahoo.de](mailto:jankorichter@yahoo.de)>`)。参阅 http://www.sai.msu.su/~megera/postgres/gist/ 获取额外信息。

F.6. chkpass

该模块实现用于存储加密密码的数据类型 `chkpass` 。 每个密码自动转换为加密进入，并且总是存储加密的。 为了比较， 简单比较清晰的文本密码，并且在比较之前比较函数将对它加密。

如果密码容易被破解，那么代码中有规定可以报告错误。 然而，目前只是stub，什么都不做。

如果输入字符串前面加冒号， 它被认为是一个已加密密码，没有进一步加密存储。 这允许先前加密密码进入。

在输出上，冒号在前。这可能备份并且重载没有重新加密的密码。 如果你想要不带冒号的加密密码，那么使用 `raw()` 函数。 这允许你使用带有类似Apache的 `Auth_PostgreSQL` 模块的类型。

加密使用标准的Unix函数 `crypt()` ， 所以它会遭受该函数所有常见局限性； 值得注意的是，只考虑前八个字符的密码。

注意 `chkpass` 数据类型是不可索引的。

示例用法：

```
test=# create table test (p chkpass);
CREATE TABLE
test=# insert into test values ('hello');
INSERT 0 1
test=# select * from test;
       p
----------------
 :dVGkpXdOrE3ko
(1 row)

test=# select raw(p) from test;
      raw
---------------
 dVGkpXdOrE3ko
(1 row)

test=# select p = 'hello' from test;
 ?column?
----------
 t
(1 row)

test=# select p = 'goodbye' from test;
 ?column?
----------
 f
(1 row)
```

F.6.1. 作者

D'Arcy J.M. Cain (`<[darcy@druid.net](mailto:darcy@druid.net)>`)

F.7. citext

`citext` 模块提供不区分大小写字符串类型， `citext` 。 从本质上讲，当比较值时，它内部调用 `lower` 。 否则，它的操作很像 `text` 。

F.7.1. 基本原理

当比较值时，在PostgreSQL中执行不区分大小写匹配的标准方法曾使用 `lower` 函数，比如

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

这个执行的相当好，但有一些缺点：

- 它使你的SQL语句冗长，并且你总是在列和查询值上使用 `lower` 。
- 它不使用索引，除非你使用 `lower` 创建一个函数索引。
- 如果你声明列为 `UNIQUE` 或者 `PRIMARY KEY` ， 隐式产生的索引是大小写敏感的。 因此对不区分大小写搜索无用，并且它不会不区分大小写。

`citext` 数据类型允许你在SQL查询中删除调用 `lower` ， 并且允许主键不区分大小写。 `citext` 是区域意识， 就像 `text` ， 这意味着大写字母和小写字母字符的匹配 依赖于数据库的 `LC_CTYPE` 设置规则。 另外，这种操作与查询中 `lower` 的使用是相同的。 但是因为它通过数据类型透明地完成， 你无须记得在你的查询中执行任何特别的。

F.7.2. 如何使用它

这是用法的一个简单例子：

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  md5(random()::text) );
INSERT INTO users VALUES ( 'Tom',    md5(random()::text) );
INSERT INTO users VALUES ( 'Damian', md5(random()::text) );
INSERT INTO users VALUES ( 'NEAL',   md5(random()::text) );
INSERT INTO users VALUES ( 'Bjørn',  md5(random()::text) );

SELECT * FROM users WHERE nick = 'Larry';
```

`SELECT` 语句将返回一个元组， 尽管 `nick` 列被设置为 `larry` ， 并且查询为 `Larry` 。

F.7.3. 字符串比较操作

`citext` 通过转换每个字符串到小写执行比较 （尽管调用 `lower` ） 并且然后通常比较结果。因此，比如， 考虑两个字符串相等，如果 `lower` 为了它们可能产生相同结果。

为了尽可能地模拟不区分大小写排序规则， 有一些字符串处理操作符和函数的 `citext` 特定版本。 所以，比如，当应用于 `citext` :他们不区分大小写匹配， 正则表达式运算符 `~` 和 `~*` 表现出相同操作。 对于 `!~` and `!~*` 以及 `LIKE` 运算符 `~~` 和 `~~*` 和 `!~~` 和 `!~~*` 同样是真，如果你想要匹配 大小写敏感，你可以投射运算符的参数给 `text` 。

类似地，如果它们参数是 `citext` ，所有下面的函数执行不区分大小写匹配：

- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

对于正则表达式函数， 如果你想要匹配大小写敏感， 你可以声明"c"标记以 强迫大小写匹配。否则，如果你想要大小写敏感操作， 你必须在使用这些函数之一前投射到 `text` 。

F.7.4. 限制

- `citext` 的折叠操作依赖于 你的数据库的 `LC_CTYPE` 设置。当创建数据库时， 决定如何比较值。通过Unicode标准定义 的术语中不是真的大小写不敏感。 实际上，这意味着， 只要你对你的排序规则满意， 你应该对 `citext` 的比较感到满意。 但是如果你有数据以不同语言存储在数据库中， 如果排序规则为另外一种语言， 那么一种语言的用户可能发现查询结果不如预期。
- 作为PostgreSQL 9.1， 你可以附属 `COLLATE` 规范到 `citext` 列或者数据值。 当比较折叠字符串时， `citext` 运算符将接受非缺省 `COLLATE` 规范， 但小写的起初折叠 总是按照数据库的 `LC_CTYPE` 设置被执行 （即，即使给定 `COLLATE "default"` ）。 这可能在未来版本中被改变，因此这两个步骤遵循输入 `COLLATE` 规范。
- `citext` 不像 `text` 一样有效， 因为运算符函数和B树比较函数必须开始数据拷贝 ， 并且为了比较将它转换为小写。然而， 它比起使用 `lower` 获取大小写不敏感匹配 更加有效。

- 如果你需要数据比较某些情况中 大小写敏感和其他情况中大小写不敏感， `citext` 没有太大帮助。 当你需要不区分大小写比较时， 标准答案是使用 `text` 类型 并且手动使用 `lower` 函数。 如果很少需要不区分大小写比较， 那么它执行正确。 如果你需要不区分大小写行为大多数时间并且很少不区分大小写， 当你想要区分大小写比较时， 考虑存储数据为 `citext` 并且明确投射该列到 `text` 。 如果你想要快速搜索的两个类型， 在这两种情况下，你将需要两个索引。
- 包含 `citext` 运算符的模式 必须在当前的 `search_path` (通常 `public`)中; 如果不是，相反调用正常的大小写敏感 `text` 运算符。

F.7.5. 作者

David E. Wheeler `<[david@kineticode.com](mailto:david@kineticode.com)>`

灵感来源于Donald Fraser的最初的 `citext` 模块。

F.8. cube

这个模块为了表示多维立方体实现了数据类型 `cube` 。

F.8.1. 语法

Table F-1为 `cube` 类型显示有效外部表示。 `_x_` , `_y_` 等表示浮点数。

Table F-1. Cube外部表示

`_x_`	一维点（或者，零长度一维间隔）
`(``_x_`)	同上
`_x1_` , `_x2_` ,..., `_xn_`	n维空间点，内部表示为零体积立方体
`(``_x1_` , `_x2_` ,..., `_xn_`)	同上
`(``_x_` ),( `_y_`)	一维间隔起始于 `_x_` 并且结束于 `_y_` 或者反之亦然；顺序并不重要
`[(``_x_` ),( `_y_`)]	同上
`(``_x1_` ,..., `_xn_` ), ( `_y1_` ,..., `_yn_`)	一个n维立方体通过一对斜对角线地对立角表示
`[(``_x1_` ,..., `_xn_` ), ( `_y1_` ,..., `_yn_`)]	同上

它无关立方体的相对角进入的哪个顺序。 如果需要创建一个统一的"左下 — 右上"内部表示，该 `cube` 函数自动交换值。

忽略空格，因此 `[(``_x_` ),( `_y_` )]和 `[(` `_x_` ), ( `_y_`)]是一样的。

F.8.2. 精确度

值内部被存储为64位浮点数。 这意味着超过16位有效数字将被截断。

F.8.3. 用法

`cube` 模块包含 `cube` 值的GiST索引操作类。 通过GiST操作符类支持的操作符显示在Table F-2中。

Table F-2. Cube GiST 运算符

运算符	描述
a = b	立方体a和b是相同的。
a && b	立方体a和b重叠
a @> b	立方体a包含立方体b。
a <@ b	立方体a被包含在立方体b中。

(PostgreSQL 8.2之前， 包含操作符 @> 和 <@ 分别称为 @ 和 ~ 。 这些名字仍然可用，但是被否决并且最终被废弃。 请注意旧的名称从先前遵循核心几何数据类型的规定中被反转！)

提供标准B树运算符，比如

运算符	描述
[a, b] < [c, d]	小于
[a, b] > [c, d]	大于

这些操作符没有任何实际目的意义但排序。 这些操作符首先比较(a)和(c)， 如果是相等的，那么比较(b)和(d)。 导致在某些情况下合理排序， 如果你想使用这种类型的ORDER BY，那么它是有用的。

Table F-3显示可用函数。

Table F-3. Cube函数

cube(float8)返回cube	两个坐标相同的一维立方体。 **cube(1) =**
cube(float8, float8)返回cube	一维立方体。 cube(1,2) == '(1),(2)'
cube(float8[])返回cube	使用数组定义的坐标的零体积立方体。
cube(float8[], float8[])返回cube	通过两个数组定义的右上和左下坐标的立 cube('{1,2}'::float[], '{3,4}'::float[
cube(cube, float8)返回cube	通过添加一个维度到新坐标两个部分相同 对于逐渐地从计算值建构立方体是有用的
cube(cube, float8, float8)返回cube	通过添加一个维度到已存在立方体上构建 构立方体是有用的。 cube('(1,2)',3,4)
cube_dim(cube)返回int	返回立方体的维数
cube_ll_coord(cube, int)返回double	返回立方体左下角的第n个坐标值
cube_ur_coord(cube, int)返回double	返回立方体右上角的第n个坐标值
cube_is_point(cube)返回bool	如果cube是一点，那么返回真。也就是说
cube_distance(cube, cube)返回double	返回两个立方体之间的距离。 如果两个立
cube_subset(cube, int[])返回cube	从一个已经存在立方体构建新的立方体，找单维的LL和UR坐标，比如， cube_subset(cube('(1,3,5),(6,7,8)'), A 度，或者按照需要重新排序它们，比如 cube_subset(cube('(1,3,5),(6,7,8)'), A
cube_union(cube, cube)返回cube	产生两个立方体并集
cube_inter(cube, cube)返回cube	产生两个立方体交集
cube_enlarge(cube c, double r, int n)返回cube	通过至少n维的指定半径增加立方体大小 围搜索临近点创建边界区域是很有用的。 了r，UR坐标增加了r。 如果LL坐标增加 发生） 比起两个坐标系被设置为平均来 增加(r >= 0)， 然后使用0作为额外坐标

F.8.4. 缺省

我相信这个联合：

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

不违背常识，也不违背交集

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

在不同的维度立方体的所有二进制运算中，我认为降低笛卡尔积投影维数，比如， 在字符串表示中忽略坐标的地方归零。上面例子等同于：

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

下面的包含谓词使用point语法， 而实际上第二个参数通过box内部表示。 这个语法没必要定义单独的point类型 以及(box,point)谓词函数。

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.8.5. 注意

对于用法实例，参阅回归测试 `sql/cube.sql` 。

为了使人们突破该事物更加难， 在立方体维度数量上有100的限制。 如果你需要大一些的，可以在 `cubedata.h` 中设置。

F.8.6. 赞扬

原作者：Gene Selkov, Jr. `<[selkovjr@mcs.anl.gov](mailto:selkovjr@mcs.anl.gov)>` ，数学和计算机科学系，Argonne国家实验室。

首先感谢 Prof. Joe Hellerstein(http://db.cs.berkeley.edu/jmh/) 阐明GiST (http://gist.cs.berkeley.edu/)的要点， 和他以前的学生Andy Dong (http://best.me.berkeley.edu/~adong/)， 比如书面说明例子，http://best.berkeley.edu/~adong/rtree/index.html。 我还要感谢所有现在的和以前的Postgres开发人员，使我可以创造我的世界并且在这个领域生存。 并且我还想要感谢Argonne Lab和能源 U.S. Department对我的数据库研究多年的忠实支持。

这个包较小更新是由Bruno Wolff III `<[bruno@wolff.to](mailto:bruno@wolff.to)>` 在2002年八月/九月完成的。 这些包含从单精度到双精度改变精度以及添加一些新的函数。

额外更新是由Joshua Reich `<[josh@root.net](mailto:josh@root.net)>` 在2006年7月进行的。 这些包含`cube(float8[], float8[])`并且 使用V1调用协议而不是过时的V0协议来清理代码。

F.9. dblink

Table of Contents

`dblink` 是一个在数据库会话中支持连接其他PostgreSQL 数据库的模块。

参阅postgres_fdw使用更现代且符合标准的基础设施 提供了大体一样的功能。

dblink_connect

Name

dblink_connect -- 打开到远程数据库的持久连接

Synopsis

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

描述

`dblink_connect()` 建立了远程PostgreSQL数据库的连接。 连接的服务器和数据库是通过标准libpq连接字符串被标识。 随意的，给连接分配名字。可以一次打开多个命名的连接，但是每次仅仅允许一个未命名连接。 连接将持续直到关闭或者数据库会话结束为止。

连接字符串也可能是已存在外服务器的名字。当定义外服务器时，推荐使用外数据包 `dblink_fdw` 。 请看下面的例子，以及CREATE SERVER和CREATE USER MAPPING。

参数

`conname`

用于连接的名称；如果忽略，那么打开一个未命名的连接， 替换任何已存在未命名连接。

`connstr`

libpq-形式连接信息字符串，比如 `hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd` 。 更多详情请参阅Section 31.1.1。 另外，外服务器名称。

返回值

返回状态总是 `OK` （因为任何错误导致函数抛出错误而不是返回）。

注意

只有超级用户可能使用 `dblink_connect` 创建非口令认证连接。如果 非超级用户需要这个能力，使用 `dblink_connect_u` 。

选择包含等号的连接名是不明智的，正如这在其他 `dblink` 函数中打开 与连接信息字符串混淆的风险。

例子

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

-- 外数据封装功能
-- 注意: 本地连接必须要求密码认证工作正常
--        否则, 你将接收到来自dblink_connect()的下面的错误:
--        ----------------------------------------------------------------------
--        ERROR: 需要密码
--        DETAIL: 如果服务器不需要密码, 那么非超级用户无法连接。
--        HINT:  必须改变目标服务器的身份验证方法。

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1', dbnam

CREATE USER dblink_regression_test WITH PASSWORD 'secret';
CREATE USER MAPPING FOR dblink_regression_test SERVER fdtest OPTIONS (user 'dblink_regres
GRANT USAGE ON FOREIGN SERVER fdtest TO dblink_regression_test;
GRANT SELECT ON TABLE foo TO dblink_regression_test;

\set ORIGINAL_USER :USER
\c - dblink_regression_test
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn','SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM dblink_regression_test;
REVOKE SELECT ON TABLE foo FROM dblink_regression_test;
DROP USER MAPPING FOR dblink_regression_test SERVER fdtest;
DROP USER dblink_regression_test;
DROP SERVER fdtest;
```

dblink_connect_u

Name

dblink_connect_u -- 危险的打开一个到远程数据库的持久连接

Synopsis

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

描述

`dblink_connect_u()` 与 `dblink_connect()` 是相同的， 除了它将允许非超级用户使用任何身份验证方法进行连接。

如果远程服务器选择一个不涉及密码的身份验证方法，然后模拟并且产生权限的随后升级，因为该会话将出现于用户，正如本地PostgreSQL服务器运行的那个。 同时，即使远程服务器确实需要密码，从服务器环境提供密码是可能的，比如 `~/.pgpass` 文件从属于服务器的用户。这不仅仅打开模拟风险，而且 可能将密码暴露给不信任的远程服务器。 因此， `dblink_connect_u()` 初始安装从 `PUBLIC` 撤销的所有权限。 使得它不可请求即付只有超级用户可以。 在一些情况下可以为 `dblink_connect_u()` 指定可以信赖的用户适当授予 `EXECUTE` 权限，但这应该仔细的做。 推荐任何从属于服务器的用户的 `~/.pgpass` 文件不包含指定通配符主机名的任何记录。

更多详情请参阅 `dblink_connect()` 。

dblink_disconnect

Name

dblink_disconnect -- 关闭远程数据库的持久连接

Synopsis

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

描述

`dblink_disconnect()` 关闭先前通过 `dblink_connect()` 打开的连接。无参数形式关闭一个未命名连接。

参数

`conname`

被关闭的命名连接的名称。

返回值

返回状态始终是 `OK` （因为任何错误导致函数抛出错误而不是返回）。

例子

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

Name

dblink -- 在远程数据库中执行查询

Synopsis

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

描述

`dblink` 在远程数据库中执行查询（通常 `SELECT` ，但是它 可以是任何返回行的SQL语句）。

当给定两个 `text` 参数时，第一个作为持久连接名称首先被查找；如果发现了， 那么在连接上执行该命令。如果没有发现，那么第一个参数被看作 `dblink_connect` 的连接信息字符串，并且指定连接为了该命令的整个时间段。

参数

`conname`

要使用的连接名称；省略这个参数使用未命名连接。

`connstr`

连接信息字符串，如先前描述的 `dblink_connect` 。

`sql`

你希望在远程数据库中执行的SQL查询，比如 `select * from foo` 。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数没有返回行。

返回值

该函数返回查询产生的行。因为`dblink`可以用于任何查询， 它被声明为返回`record`，而不是指定任何特定列。这意味着 在调用查询中你必须指定预期列； 否则PostgreSQL不知道发生什么，这有个例子：

```
SELECT *
    FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

`FROM`子句的"alias"部分必须指定函数将要返回的列名和类型。 （在别名中指定列名实际上是标准SQL语法，但是指定列类型是PostgreSQL扩展。） 这允许系统理解`*`应该扩展到什么，以及在`WHERE`子句中`proname` 指向什么，提前尝试执行该函数。在运行时，如果远程数据库的实际查询结果没有`FROM`子句中显示 的相同列数，那么将抛出错误。列名不需要匹配，然而，`dblink`不坚持确切类型匹配。只要返回的数据字符串对于`FROM`子句中声明的 列类型是有效输入那么它将成功。

注意

预定义查询使用`dblink`的一个便捷方式是创建视图。 这允许将列类型信息放在视图中，而不是在每个查询中拼出它。比如，

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

例子

```
SELECT * FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

Name

dblink_exec -- 在远程数据库中执行命令

Synopsis

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

描述

`dblink_exec` 在远程数据库中执行命令（即不返回行的任何SQL语句）。

当给定两个 `text` 参数时，第一个作为持久连接名称首先被查找；如果发现了， 那么在连接上执行该命令。如果没有发现， 那么第一个参数被看作 `dblink_connect` 的连接信息字符串，并且指定连接为了该命令的整个时间段。

参数

`conname`

要使用的连接名称；省略这个参数使用未命名连接。

`connstr`

连接信息字符串，如先前描述的 `dblink_connect` 。

`sql`

你希望在远程数据库中执行的SQL命令， 比如 `insert into foo values(0,'a','{"a0","b0","c0"}')` 。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数的返回值设置为 `ERROR` 。

返回值

返回状态，该命令的状态字符串或者 `ERROR` 。

例子

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21,''z'',''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

Name

dblink_open -- 打开远程数据库中的游标

Synopsis

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns text
```

描述

`dblink_open()` 打开了远程数据库中的游标。 游标可以随后使用 `dblink_fetch()` 和 `dblink_close()` 被操作。

参数

`conname`

要使用的连接名称；省略这个参数使用未命名连接。

`cursorname`

分配给这个游标的名称。

`sql`

你希望在远程数据库中执行的 `SELECT` 语句，比如 `select * from pg_class` 。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数的返回值设置为 `ERROR` 。

返回值

返回状态， `OK` 或者 `ERROR` 。

注意

因为游标只能停留在事务块中，如果远程端已经不在事务中， 那么 `dblink_open` 在远程端开始显式事务块（ `BEGIN` ）， 当执行匹配的 `dblink_close` 时，则该事务将再次被关闭。 注意如果在 `dblink_open` 和 `dblink_close` 之间使用 `dblink_exec` 改变数据，那么会产生错误或者你在 `dblink_close` 之前使用 `dblink_disconnect` ，你的改变将丢失，因为终止了事务。

例子

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

Name

dblink_fetch -- 从远程数据库中打开的游标中返回行

Synopsis

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns
```

描述

`dblink_fetch` 通过 `dblink_open` 预先建立的游标中抓取行。

参数

`conname`

要使用的连接名称；省略这个参数使用未命名连接。

`cursorname`

获取游标名称。

`howmany`

要检索的最大行数。抓取下一个 `howmany` 行， 从当前光标位置开始，向前移动。一旦游标已经到达末尾，不会产生更多行。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数没有返回行。

返回值

该函数返回从游标中抓取的行。要使用该函数，你将需要 指定预期的字段集，正如前面讨论的 `dblink` 。

注意

在 `FROM` 子句上指定的返回列数之间的不匹配，并且通过远程游标返回实际列数，将抛出一个错误。在这种情况下，远程游标仍然按照没有产生错误时一样的行增长。 同样的在远程 `FETCH` 执行后本地查询产生的任何其他错误也是这样。

例子

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like ''bytea
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

Name

dblink_close -- 关闭远程数据库中的游标

Synopsis

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

描述

`dblink_close` 关闭先前使用 `dblink_open` 打开的游标。

参数

`conname`

要使用的连接名称；省略这个参数使用未命名连接。

`cursorname`

关闭的游标名称。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数的返回值被设置为 `ERROR` 。

返回值

返回状态， `OK` 或者 `ERROR` 。

注意

如果 `dblink_open` 启动了一个显式事务块，并且这是在该连接中最后保持打开的游标。 `dblink_close` 将提交匹配的 `COMMIT` 。

例子

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

Name

dblink_get_connections -- 返回所有打开命名dblink连接的名字

Synopsis

```
dblink_get_connections() returns text[]
```

描述

`dblink_get_connections` 返回所有打开已命名 `dblink` 连接的名字数组。

返回值

返回连接名称的文本数组，如果没有的话返回NULL。

例子

```
SELECT dblink_get_connections();
```

dblink_error_message

Name

dblink_error_message -- 获取命名连接上的最后错误信息

Synopsis

```
dblink_error_message(text connname) returns text
```

描述

`dblink_error_message` 抓取给定连接的最新远程错误信息。

参数

`conname`

使用连接的名字。

返回值

返回最后错误信息，或者如果在这个连接中没有错误，则返回空字符串。

例子

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

Name

dblink_send_query -- 发送一个异步查询到远程数据库

Synopsis

```
dblink_send_query(text connname, text sql) returns int
```

描述

`dblink_send_query` 发送异步执行的查询，也就是说，没有立即等待结果。 在连接进行的过程中没有一个异步查询。

在成功调度异步查询之后，完成状态可以用 `dblink_is_busy` 被检查， 结果最终使用 `dblink_get_result` 收集。 尝试使用 `dblink_cancel_query` 取消活跃的异步查询是可能的。

参数

`conname`

使用的连接名字。

`sql`

你希望在远程数据库中执行的SQL语句，比如 `select * from pg_class` 。

返回值

如果查询被成功调度，则返回1。否则返回0。

例子

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

Name

dblink_is_busy -- 检查是否连接忙于异步查询

Synopsis

```
dblink_is_busy(text connname) returns int
```

描述

`dblink_is_busy` 测试是否异步查询在进行中。

参数

`conname`

检查连接的名字。

返回值

如果连接繁忙，则返回1，如果不繁忙，则返回0。 如果该函数返回0，那么保证 `dblink_get_result` 将不阻塞。

例子

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

Name

dblink_get_notify -- 在连接上检索异步通知

Synopsis

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra text)
```

描述

`dblink_get_notify`在未命名连接上检索通知，或者如果指定了则在命名的连接上。 为了通过dblink接收通知，必须首先使用`dblink_exec`发出`LISTEN`， 更多详情请参阅LISTEN和NOTIFY。

参数

`conname`

获取通知的已命名连接名字。

返回值

返回`setof (notify_name text, be_pid int, extra text)`， 如果没有则返回空集。

例子

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

Name

dblink_get_result -- 获得异步查询结果

Synopsis

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

描述

`dblink_get_result` 收集先前使用 `dblink_send_query` 发送的异步查询结果。 如果还没完成查询，那么 `dblink_get_result` 将等待它。

参数

`conname`

要使用的连接名字。

`fail_on_error`

如果真（忽略时缺省）那么在连接的远程端抛出的错误也会导致本地抛出错误， 如果假，那么远程错误在本地作为NOTICE被报告， 并且函数没有返回行。

返回值

一个异步查询（也就是说，返回行的SQL语句）， 该函数返回通过查询产生的行。要使用该函数，你将需要指定预期的字段集， 正如前面讨论的 `dblink` 。

一个异步命令（也就是说，没有返回行的SQL语句）， 该函数返回带有该命令的状态字符串的单文本列的单一行。 声明结果将在调用 `FROM` 子句中有单一文本列。

注意

如果 `dblink_send_query` 返回1，那么必须调用该函数。 它必须被每一个查询调用，并且一个额外时间来获取空集结果， 连接之前可以再次使用。

当在返回它到本地查询处理器之前使用 `dblink_send_query` 和 `dblink_get_result` , dblink抓取整个远程查询结果。如果查询返回大量行，那么 这可以导致本地会话短暂内存膨胀。它可以更好的打开这个查询比如使用 `dblink_open` 的游标， 然后抓取每次可管理的行数。另外，使用普通的 `dblink()` 通过多任务缓冲处理大量结果集到磁盘避免内存膨胀。

例子

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3; select
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

Name

dblink_cancel_query -- 取消在已经命名连接上的任何活跃查询

Synopsis

```
dblink_cancel_query(text connname) returns text
```

描述

`dblink_cancel_query` 尝试取消在已命名连接上进行的任何查询。 注意这不一定成功（因为，比如远程查询可能已经完成）。 取消查询仅仅提高了查询将失败的可能性。你还必须完成正常查询协议，比如 通过调用 `dblink_get_result` 。

参数

`conname`

使用连接名称。

返回值

如果已经发送取消请求，那么返回 `OK` ， 或者失败的错误消息文本。

例子

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

Name

dblink_get_pkey -- 返回位置和关系的主键字段的字段名字

Synopsis

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

描述

`dblink_get_pkey` 提供在本地数据库中关系的主键的信息。 有时在生成被发送到远程数据库的查询中有用。

参数

`relname`

本地关系名字，比如 `foo` 或者 `myschema.mytab` 。如果名字是混合情况下或者包含特殊字符，那么包含双引号， 比如 `"FooBar"` ；没有引号，则字符串被折叠成小写字母。

返回值

为每个主键字段返回一行，如果关系没有主键，那么不返回行。结果行类型被定义为

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

`position` 列简单的从1到 `_N_` 运行； 它是主键内的字段数，而不是表的列数。

例子

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

Name

dblink_build_sql_insert -- 使用本地元组建立INSERT语句，使用可选的已提供的值替换主键字段

Synopsis

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

描述

`dblink_build_sql_insert` 可以用于执行远程数据库本地表的选择行复制。 它从基于主键的本地表选择行，然后建立一个SQL `INSERT` 命令复制该行， 但是随着主键值被最后参数中的值替换。 （为了准确拷贝行，请为最后两个参数指定同一值）。

参数

`relname`

本地关系名称，比如 `foo` 或者 `myschema.mytab` 。如果名字是混合情况下或者包含特殊字符，那么包含双引号， 比如 `"FooBar"` ；没有引号，则字符串被折叠成小写字母。

`primary_key_attnums`

主键字段的属性数量（1维），比如 `1 2` 。

`num_primary_key_atts`

主键字段数。

`src_pk_att_vals_array`

主键字段值用于查找本地元组。每个字段用文本形式表示。 如果在这些主键值中没有本地行，那么抛出错误。

`tgt_pk_att_vals_array`

主键字段值被放置在`INSERT`命令中。每个字段用文本形式表示。

返回值

作为文本返回所请求的SQL语句。

注意

PostgreSQL 9.0， `primary_key_attnums`中 的属性号被解释为逻辑列数，对应`SELECT * FROM relname`中的列的位置。 先前版本作为物理列位置进行解释。如果在整个表周期中指定列左侧的任何列已经被删除了， 那么这是有区别的。

例子

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
             dblink_build_sql_insert
--------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

Name

dblink_build_sql_delete -- 使用已提供的值为主键字段值建立一个DELETE语句

Synopsis

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

`dblink_build_sql_delete` 可以用于执行远程数据库本地表的选择行复制。 它建立一个SQL `DELETE` 命令将删除给定主键值的行。

参数

`relname`

本地关系名字，比如 `foo` 或者 `myschema.mytab` 。如果名字是混合情况下或者包含特殊字符，那么包含双引号， 比如 `"FooBar"` ；没有引号，则字符串被折叠成小写字母。

`primary_key_attnums`

主键字段的属性数量（1维），比如 `1 2` 。

`num_primary_key_atts`

主键字段数量。

`tgt_pk_att_vals_array`

主键字段值用于 `DELETE` 命令。每个字段用文本形式表示。

返回值

作为文本返回所请求的SQL语句。

注意

PostgreSQL 9.0， `primary_key_attnums` 中 的属性号被解释为逻辑列数，对应 `SELECT * FROM relname` 中的列的位置。 先前版本作为物理列位置进行解释。如果在整个表周期中指定列左侧的任何列已经被删除了， 那么这是有区别的。

例子

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

Name

dblink_build_sql_update -- 使用本地元组建立UPDATE语句， 使用可选的已提供的值替换主键字段

Synopsis

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

描述

`dblink_build_sql_update` 可以用于执行远程数据库本地表的选择行复制。 它从基于主键的本地表选择行，然后建立一个SQL `UPDATE` 命令复制该行， 但是随着主键值被最后参数中的值替换。 （为了准确拷贝行，请为最后两个参数指定同一值）。 `UPDATE` 命令总是 分配行中所有字段；在这个和 `dblink_build_sql_insert` 之间的 主要区别是目标行已经存在于远程表中是个假设。

参数

`relname`

本地关系名字，比如 `foo` 或者 `myschema.mytab` 。如果名字是混合情况下或者包含特殊字符，那么包含双引号， 比如 `"FooBar"` ；没有引号，则字符串被折叠成小写字母。

`primary_key_attnums`

主键字段的属性数量（1维），比如 `1 2` 。

`num_primary_key_atts`

主键字段数量。

`src_pk_att_vals_array`

主键字段值用于查找本地元组。每个字段用文本形式表示。 如果在这些主键值中没有本地行，那么抛出错误。

`tgt_pk_att_vals_array`

主键字段值被放置在 `UPDATE` 命令中。每个字段用文本形式表示。

返回值

作为文本返回请求的SQL语句。

注意

PostgreSQL 9.0， `primary_key_attnums` 中 的属性号被解释为逻辑列数，对应 `SELECT * FROM relname` 中的列的位置。 先前版本作为物理列位置进行解释。如果在整个表周期中指定列左侧的任何列已经被删除了， 那么这是有区别的。

例子

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.10. dict_int

`dict_int` 是一个全文检索的扩展字典模板的示例。这个示例字典的动机是控制整数（有符号和无符号）的索引，允许在防止唯一码的数量过度增长时给整数索引，这将大大的影响搜索的性能。

F.10.1. 配置

该字典接受两个选项：

- `maxlen` 参数声明整数数字的最大位数。缺省值是6。
- `rejectlong` 参数声明一个超长的整数是否应该被截断或忽略。如果 `rejectlong` 为 `false`（缺省），那么字典返回该整数前面的 `maxlen` 位。如果 `rejectlong` 为 `true`，那么字典认为一个超长的整数是结束词，所以它将不会被索引。请注意，这也意味着这样一个整数不能被搜索。

F.10.2. 用法

安装 `dict_int` 扩展创建基于它的一个文本搜索模板 `intdict_template` 和一个字典 `intdict`，带有缺省的参数。你可以修改这些参数，例如

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

或创建新的基于该模板的字典。

要测试这个字典，可以尝试

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

但是真实的使用情况将包括包含它在一个文本搜索配置里，就像Chapter 12 里描述的那样。可能看起来像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.11. dict_xsyn

`dict_xsyn`（扩展的同义词词典）是一个全文本搜索的扩展字典模板的例子。 这个字典类型用它们的同义词组替代单词，并且因此使得使用任意它的同义词搜索这个单词成为可能。

F.11.1. 配置

`dict_xsyn`字典接受下列的选项：

- `matchorig`控制字典是否接受原始单词。缺省是`true`。
- `matchsynonyms`控制字典是否接受同义词。缺省是`false`。
- `keeporig`控制原始单词是否包含在字典的输出里。缺省是`true`。
- `keepsynonyms`控制同义词是否包含在字典的输出里。缺省是`true`。
- `rules`是包含同义词列表的文件的基名。这个文件必须存储在`$SHAREDIR/tsearch_data/`（这里的`$SHAREDIR`意思是PostgreSQL安装的共享数据目录）里。 它的名字必须以`.rules`（不包含在`rules`参数中）结束。

rules文件有下列的格式：

- 每行代表着一个单词的同义词组，单词在每行的开头给出。同义词用空格分开：
- 符号井号(`#`)是一个注释分隔符。它可能在一行中的任意位置出现。该行在它后面的部分将被跳过。

查看`xsyn_sample.rules`（安装在`$SHAREDIR/tsearch_data/`里）获取一个实例。

F.11.2. 用法

安装`dict_xsyn`扩展创建基于它的一个文本搜索模板`xsyn_template`和一个字典`xsyn`，带有缺省的参数。你可以修改这些参数，如

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

或创建新的基于该模板的字典。

要测试该字典，你可以尝试

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false, MATCHSYNONYMS=
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true, MATCHORIG=false
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

实际的使用情况包括包含它在一个文本搜索配置（在Chapter 12里描述）里。 可能看起来像这样：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.12. dummy_seclabel

`dummy_seclabel` 模块的存在只是为了支持 `SECURITY LABEL` 语句的回归测试。 不是为了在生产中使用。

F.12.1. 基本原理

`SECURITY LABEL` 语句允许用户给数据库对象分配安全标签；不过， 安全标签只有当可加载模块特别允许时才能分配，所以这个模块提供来允许合适的回归测试。

打算将安全标签提供者用于生产中将典型的取决于特定于平台的特征，如SE-Linux。 这个模块是依赖于平台的，并且因此很好的适合于回归测试。

F.12.2. 用法

这里是使用的一个例子：

```
# postgresql.conf
shared_preload_libraries = 'dummy_seclabel'
```

```
postgres=# CREATE TABLE t (a int, b text);
CREATE TABLE
postgres=# SECURITY LABEL ON TABLE t IS 'classified';
SECURITY LABEL
```

`dummy_seclabel` 模块只提供四种硬编码标签： `unclassified` , `classified` , `secret` , and `top secret` 。 不允许任何其他的字符串作为安全标签。

这些标签不是用来强制访问控制的。它们只是用来检查 `SECURITY LABEL` 语句是否像预期的那样工作。

F.12.3. 作者

KaiGai Kohei `<[kaigai@ak.jp.nec.com](mailto:kaigai@ak.jp.nec.com)>`

F.13. earthdistance

`earthdistance`模块提供两个不同的方法计算地球表面上的大圆弧距离。 描述的第一个依赖于 `cube` 模块（必须在 `earthdistance` 可以安装之前安装）。第二个基于内建的 `point` 数据类型， 使用经线和纬线作为坐标。

在这个模块，假设地球为完美的球形。（如果对你来说太不准确，你可能想查阅 PostGIS项目。）

F.13.1. 基于立方体的地球距离

数据存储在立方体中，点用3个坐标表示从地球中心到表面的x, y, z的距离。 提供一个在 `cube` 之上的域 `earth` ，包含检查约束， 值必须符合限制条件并且合理的接近地球的实际表面。

地球的半径从 `earth()` 函数获得。以米给出。 可以通过改变模块使用其他单元来改变这个函数，或者使用你觉得更合适的半径值。

这个包也应用到了天文数据库。天文学家可能想要改变 `earth()` ， 使其返回一个 `180/pi()` 的半径，这样距离可以用角度表示。

提供函数支持经纬度（以角度方式）输入、输出、计算两个点之间大圆弧的距离， 和容易的指定用于索引查询的边界框。

提供的函数在Table F-4里显示。

Table F-4. 基于立方体的地球距离函数

函数	返回	描述
`earth()`	`float8`	返回地球的假设半径。
`sec_to_gc(float8)`	`float8`	将地球表面两点之间正常的直线（割线）距离转换为大圆弧的距离。
`gc_to_sec(float8)`	`float8`	将地球表面两点之间的大圆弧的距离转换为正常的直线（割线）距离。
`ll_to_earth(float8, float8)`	`earth`	以角度形式返回给定纬度（参数 1）和经度（参数 2）的地球表面点的位置。
`latitude(earth)`	`float8`	以角度形式返回地球表面一个点的纬度。
`longitude(earth)`	`float8`	以角度形式返回地球表面一个点的经度。
`earth_distance(earth, earth)`	`float8`	返回地球表面两点之间的大圆弧距离。
`earth_box(earth, float8)`	`cube`	返回一个适合于索引搜索的立方体，该立方体在一个给定位置的大圆弧距离里使用点的立方 `@>` 操作符。在这个立方体中的某些点可能比指定的大圆弧距离的位置更远， 所以使用 `earth_distance` 的第二个检查应该包含在查询中。

F.13.2. 基于点的地球距离

模块的第二部分依赖于用类型为 `point` 的值表示地球距离， 这里第一个组成部分用来表示以角度表示的经度，第二个组成部分用来表示以角度表示的纬度。 点被当做是(经度, 纬度)并且反过来不行，因为经度为X轴纬度为Y轴。

提供了一个操作符，在Table F-5中显示。

Table F-5. 基于点的地球距离操作符

操作符	返回	描述
`point` `<@>` `point`	`float8`	给出地球表面两点之间的法定英里距离。

请注意，不像基于 `cube` 部分的模块，这里的单元是硬链接的： 改变 `earth()` 函数将不会影响这个操作符的结果。

经度/纬度表示法的一个缺点是：你必须注意靠近两极和接近+/- 180度经线的边界条件。 基于 `cube` 的表示法避免了这些间断点。

F.14. file_fdw

`file_fdw` 模块提供了外部数据封装器 `file_fdw`， 可以用来在服务器的文件系统中访问数据文件。数据文件必须是 `COPY FROM` 可读的格式；参阅COPY获取细节。访问这样的数据文件当前只是可读的。

使用这个封装器创建的外部表可以有下列选项：

`filename`

指定要读取的文件。这是必需的。必须是一个绝对路径名。

`format`

指定文件的格式，与 `COPY` 的 `FORMAT` 选项相同。

`header`

指定文件是否有标题行，与 `COPY` 的 `HEADER` 选项相同。

`delimiter`

指定文件的分隔符，与 `COPY` 的 `DELIMITER` 选项相同。

`quote`

指定文件的引用字符，与 `COPY` 的 `QUOTE` 选项相同。

`escape`

指定文件的逃逸字符，与 `COPY` 的 `ESCAPE` 选项相同。

`null`

指定文件的null字符串，与 `COPY` 的 `NULL` 选项相同。

`encoding`

指定文件的编码，与 `COPY` 的 `ENCODING` 选项相同。

请注意，当 `COPY` 允许的选项如OIDS和HEADER不带有相应的值被声明时， 外部数据封装器语法在所有的情况下都需要一个值。要激活 `COPY` 选项通常不提供值，不过你可以传递值TRUE。

用这个触发器创建的外部表的一个字段可以有下列的选项：

`force_not_null`

这是一个布尔选项。如果为真，则声明字段的值不应该匹配空字符串（也就是， 文件级别 `null` 选项）。这与列出 `COPY` 的 `FORCE_NOT_NULL` 选项里的字段有相同的效果。

`file_fdw` 目前不支持 `COPY` 的 `OIDS` 和 `FORCE_QUOTE` 选项。

这些选项只能为外部表或它的字段声明，不是在 `file_fdw` 外部数据封装器的选项里， 也不是在使用该封装器的服务器或用户映射的选项里。

修改表级别的选项需要超级用户权限，因为安全原因：只有超级用户能够决定读哪个文件。原则上非超级用户可以被允许改变其他选项，但是目前还不支持。

对于一个使用 `file_fdw` 的外部表， `EXPLAIN` 显示要读取的文件名。 除非指定了 `COSTS OFF` ，否则也显示文件大小（字节计）。

Example F-1. 为PostgreSQL CSV日志创建一个外部表

`file_fdw` 明显的用处之一就是使PostgreSQL活动日志可以作为一个表查询。 要做到这点，首先必须登录到一个CSV文件，这里我们称为 `pglog.csv` 。 首先，作为一个扩展安装 `file_fdw` 。

```
CREATE EXTENSION file_fdw;
```

然后创建一个外部服务器：

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

现在已经准备好了创建外部数据表。使用 `CREATE FOREIGN TABLE` 命令， 需要为表定义字段、CSV文件名和它的格式：

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/9.1/data/pg_log/pglog.csv', format 'csv' );
```

就这样，现在可以查询日志目录。当然，在生产中需要定义一些处理日志回旋的方法。

F.15. fuzzystrmatch

`fuzzystrmatch`模块提供几个函数判断字符串之间的相似点和距离。

Caution

目前，`soundex`，`metaphone`，`dmetaphone`，和`dmetaphone_alt`不适合多字节编码（如UTF-8）。

F.15.1. Soundex

Soundex 系统是一种通过转换为相同代码匹配相似发音名字的方法。 最初是在1880、1900和1910年用于美国的人口普查。注意Soundex对于非英文的名字不是很有帮助。

`fuzzystrmatch`模块提供和Soundex代码一起使用的两个函数:

```
soundex(text) returns text
difference(text, text) returns int
```

`soundex`函数转换字符串为Soundex代码。`difference` 函数转换两个字符串为它们的Soundex代码然后报告匹配代码位置的数量。 因为Soundex代码有四个字符，结果范围从零到四，零表示没有匹配而四表示完全匹配。 （因此，这个函数取名不当，`similarity`本是一个好名字。）

这里是一些有用的示例：

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.15.2. Levenshtein

这个函数计算两个字符串之间的Levenshtein距离：

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns i
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int sub_cost
levenshtein_less_equal(text source, text target, int max_d) returns int
```

`source` 和 `target` 都可以是任意非空字符串，最多为255字节。 cost参数声明对于一个字符插入、删除或替换分别改变多少。可以省略cost参数， 就像该函数的第二种语法；这种情况下，他们缺省都是1。 `levenshtein_less_equal` 对于低距离是levenshtein函数的快速版本。如果实际距离小于或等于max_d， 那么 `levenshtein_less_equal` 返回实际值。否则这个函数返回大于max_d的值。

示例：

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.15.3. Metaphone

Metaphone， 类似于Soundex，基于为输入字符串构造一个典型的代码的想法。 如果两个字符串有相同的代码那么认为它们相似。

这个函数计算输入字符串的metaphone代码：

```
metaphone(text source, int max_output_length) returns text
```

`source` 必须为一个非空字符串，最多为255个字符。 `max_output_length` 设置输出metaphone代码的最大长度； 如果实际比这长，那么输出截断为这个长度。

示例：

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.15.4. Double Metaphone

Double Metaphone系统为一个给定的输入字符串计算两个"听起来像"的字符串— 一个"原先的"和一个"替换的"。大多数情况下它们相同， 但是取决于发音它们对于非英文名有一点不同。这些函数计算原先的和替换的代码：

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

这里对于输入字符串没有长度限制。

示例：

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.16. hstore

这个模块实现了 `hstore` 数据类型，在单个PostgreSQL值中存储一组键/值对。 这在不同的场景中是有用的，如很少检查带有许多属性的行，或半结构化的数据。 键和值是简单的文本字符串。

F.16.1. `hstore` 外部表示

`hstore` 的文本表示用于输入和输出，包括零个或更多由逗号分开的 `_key_` `=>` `_value_` 对。一些例子：

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

键/值对的顺序不重要（可能不在输出中复制）。忽略对和 `=>` 符号周围的空格。 包含空格、逗号、 `=` 或 `>` 的键和值要加双引号。 要在键或值中包含一个双引号或反斜杠，要用反斜杠逃逸。

`hstore` 中的每个键都是唯一的。如果你用重复键声明一个 `hstore` ， 将只有一个存储在 `hstore` 中，并且不保证会保存哪一个：

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

一个值（不是一个键）可以是SQL `NULL` 。如：

```
key => NULL
```

`NULL` 关键字是大小写敏感的。要将 `NULL` 当做普通的字符串来对待就要给 `NULL` 加双引号。

> **Note:** 记住 `hstore` 文本格式，当用于输入时，在任何请求的引用或逃逸之前应用。 如果通过一个参数传递一个 `hstore` 文本，那么不需要额外的处理。 但是如果作为引用的文本常量传递，那么任何单引号字符和（依赖于 `standard_conforming_strings` 配置参数的设置）反斜杠字符需要正确的逃逸。 参阅Section 4.1.2.1获取更多处理字符串常量的信息。

在输出时，键和值总是包含在双引号中，即使并不严格需要也是这样。

F.16.2. `hstore` 操作符和函数

`hstore` 模块提供的操作符显示在Table F-6中， 函数在Table F-7中。

Table F-6. `hstore` 操作符

操作符	描述	示例
hstore -> text	获得键的值(如果不存在为 NULL)	'a=>x, b=>y'::hstore -> 'a'
hstore -> text[]	获得多个键的值(如果不存在为 NULL)	'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a
hstore || hstore	连接 hstore	'a=>b, c=>d'::hstore || 'c=>x, d=&g
hstore ? text	hstore 包含键吗?	'a=>1'::hstore ? 'a'
hstore ?& text[]	hstore 包含所有指定的键?	'a=>1,b=>2'::hstore ?& ARRAY['a','b']
hstore ?| text[]	hstore 包含任何指定的键?	'a=>1,b=>2'::hstore ?| ARRAY['b','c']
hstore @> hstore	左操作符包含右操作符?	'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'
hstore <@ hstore	左操作符包含于右操作符?	'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'
hstore - text	从左操作符中删除键	'a=>1, b=>2, c=>3'::hstore - 'b'::text
hstore - text[]	从左操作符中删除键	'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']
hstore - hstore	从左操作符中删除匹配对	'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>
record #= hstore	用 hstore 里匹配的值替换 record 里的字段	查看示例章节
%% hstore	转换 hstore 为替换键和值的数组	%% 'a=>foo, b=>bar'::hstore
%# hstore	转换 hstore 为两维键/值数组	%# 'a=>foo, b=>bar'::hstore

> **Note:** PostgreSQL 8.2之前，包含操作符 @> 和 <@ 分别被称为 `@` 和 `~`。这些名字现在仍然可用，但是已经废弃了并且最终将会被移除。 请注意，旧的名字从大会移除，之前跟随着核心几何数据类型！

Table F-7. `hstore` 函数

函数	返回类型	描述	
`hstore(record)`	`hstore`	从一个记录或行构造一个 `hstore`	hst
`hstore(text[])`	`hstore`	从一个数组构造一个 `hstore`，可能是一个键/值数组，也可能是一个两维数组	hst
`hstore(text[], text[])`	`hstore`	从一个单独的键和值数组构造一个 `hstore`	hst
`hstore(text, text)`	`hstore`	制作单一项 `hstore`	hst
`akeys(hstore)`	`text[]`	获取 `hstore` 的键作为一个数组	ake
`skeys(hstore)`	`setof text`	获取 `hstore` 的键作为一个集合	ske
`avals(hstore)`	`text[]`	获取 `hstore` 的值作为一个数组	ava
`svals(hstore)`	`setof text`	获取 `hstore` 的值作为一个集合	sva
`hstore_to_array(hstore)`	`text[]`	获取 `hstore` 的键和值作为一个键值交替的数组	hst
		获	

hstore_to_matrix(hstore)	text[]	取 hstore 的键和值作为一个两维数组	hst
hstore_to_json(hstore)	json	获取 hstore 作为一个 json 值	hst
hstore_to_json_loose(hstore)	json	获取 hstore 作为一个 json 值，但是试图区分数值和布尔值，所以它们在JSON中没有引号	hst
slice(hstore, text[])	hstore	提取 hstore 的一个子集	sli
each(hstore)	setof(key text, value text)	获取 hstore 的键和值作为一个集合	sel
exist(hstore,text)	boolean	hstore 包含键吗？	exi
defined(hstore,text)	boolean	hstore 包含非 NULL 值的键吗？	def
delete(hstore,text)	hstore	删除匹配键的对	del
delete(hstore,text[])	hstore	删除匹配键的多个对	del
delete(hstore,hstore)	hstore	删除匹配第二个参数中元素的对	del
populate_record(record,hstore)	record	替换 record 中匹配 hstore 中的值的字段	参阅

> **Note:** 当 `hstore` 值转换为 `json` 时使用 `hstore_to_json` 函数。
>
> **Note:** 函数 `populate_record` 实际上是用 `anyelement`， 而不是 `record`， 声明为它的第一个参数，但是它将用运行时错误拒绝非记录类型。

F.16.3. 索引

`hstore` 有GiST 和 GIN索引支持 `@>`, `?`, `?&` 和 `?|` 操作符。例如：

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

`hstore` 也为 `=` 操作符支持 `btree` 或 `hash` 索引。 这允许 `hstore` 字段声明为 `UNIQUE`， 或在 `GROUP BY`, `ORDER BY` 或 `DISTINCT` 表达式中使用。为 `hstore` 值的排序顺序不是很有用，但是这些索引可能对于等价查找有用处。为 `=` 比较创建索引如下：

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.16.4. 例子

添加一个键，或用新值更新一个现有的键：

```
UPDATE tab SET h = h || hstore('c', '3');
```

删除一个键：

```
UPDATE tab SET h = delete(h, 'k1');
```

转换 `record` 为 `hstore`：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                   hstore
---------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

转换 `hstore` 为一个预先定义的 `record` 类型：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

使用 `hstore` 里的值修改现有的记录：

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.16.5. 统计

`hstore` 类型，由于其内在的慷慨，可以包含大量不同的键。检查有效的键是应用的任务。 下列的例子演示几个检查键和获取统计的技术。

简单例子：

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

使用一个表：

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

在线统计：

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
...................
```

F.16.6. 兼容性

自PostgreSQL 9.0起， `hstore`使用一个不同于以前版本的内部表示。 这样做对于转储/恢复升级没有什么障碍，因为文本表示（在转储中使用的）没有改变。

在一个二进制升级中，向上兼容是通过使新代码认识老格式的数据来维护的。 这在处理还未被新代码修改的数据时会有一点性能代偿。通过像下面这样的`UPDATE` 语句强制升级一个表字段中的所有值是可能的：

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

另一个方法是：

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

`ALTER TABLE`方法要求在表上的一个排他锁，但是不会导致有旧行版本的表膨胀。

F.16.7. 作者

Oleg Bartunov `<[oleg@sai.msu.su](mailto:oleg@sai.msu.su)>` , Moscow, Moscow University, Russia

Teodor Sigaev `<[teodor@sigaev.ru](mailto:teodor@sigaev.ru)>` , Moscow, Delta-Soft Ltd., Russia

Andrew Gierth `<[andrew@tao11.riddles.org.uk](mailto:andrew@tao11.riddles.org.uk)>` 附加的增强, United Kingdom

F.17. intagg

`intagg`模块提供一个整数聚合器和一个枚举器。 `intagg` 现在已经废弃了，因为内置的函数提供它的能力的一个超集。不过， 该模块仍然作为内置函数的兼容性封装器提供。

F.17.1. 函数

聚合器是一个生产正好包含输入的整数的整数数组的聚合函数`int_array_aggregate(integer)`。 这是`array_agg`的封装器， `array_agg`对于任意数组类型做相同的事情。

枚举器是返回`setof integer`类型的函数`int_array_enum(integer[])`。 本质上是聚合器的反向操作：给出一个整数数组，将其展开为一组行。 这是`unnest`的封装器， `unnest`对于任意数组类型做相同的事情。

F.17.2. 示例使用

许多数据库系统有一到多个表的概念。这样的一个表通常位于两个索引表之间，例如：

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

通常这样使用：

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = _item_;
```

这将返回所有在左手边的表里有记录的右手边表里的条目。这在SQL中是一个非常普通的构造。

现在，这个方法在一个有非常大数量的记录的`one_to_many`表里是很难处理的。 通常，像这样的连接将会导致索引扫描和抓取表中有左手边记录的每个右手边记录。 如果你有一个非常动态的系统，那么没有什么你可以做的。不过，如果你有一些静态的数据， 你可以使用该聚合器创建一个汇总表。

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

这将创建一个表，这个表有每个左边的条目和一个左边条目的数组。现在，如果没有使用该数组的方法则是相当无用的；这就是为什么有一个数组枚举器。你可以

```
SELECT left, int_array_enum(right) FROM summary WHERE left = _item_;
```

上面的查询使用 `int_array_enum` 产生下面相同的结果

```
SELECT left, right FROM one_to_many WHERE left = _item_;
```

不同之处是针对summary表的查询必须只从表中获取一行，而针对 `one_to_many` 的直接查询必须索引扫描然后从每条记录中获取一行。

在一个系统上，一个显示了消耗8488的查询的 `EXPLAIN` 减少到消耗329。原始查询时一个包括 `one_to_many` 表的连接，替换为:

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = _item_) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.18. intarray

`intarray`模块为操作整数的null-free数组提供一些有用的函数和操作符。 也支持使用其中的一些操作符执行索引搜索。

如果提供的数组包含任何空元素，那么所有这些操作符都将抛出一个错误。

这些操作符中的一些只对一维数组敏感。尽管他们将接受多个维数的输入数组， 数据被按照存储的顺序当做一维数组。

F.18.1. `intarray` 函数和操作符

`intarray`模块提供的函数显示在Table F-8里， 操作符显示在Table F-9里。

Table F-8. `intarray` 函数

函数	返回类型	描述	示例
`icount(int[])`	`int`	数组中元素的个数	`icount('{1,2,3}'::int[`
`sort(int[], text dir)`	`int[]`	给数组排序 — `dir` 必须为 `asc` 或 `desc`	`sort('{1,2,3}'::int[],`
`sort(int[])`	`int[]`	以升序顺序排序	`sort(array[11,77,44])`
`sort_asc(int[])`	`int[]`	以升序顺序排序	
`sort_desc(int[])`	`int[]`	以降序顺序排序	
`uniq(int[])`	`int[]`	删除相邻的重复	`uniq(sort('{1,2,3,2,1}`
`idx(int[], int item)`	`int`	匹配 `item` 的第一个元素中的索引 (如果没有则为0)	`idx(array[11,22,33,22,1`
`subarray(int[], int start, int len)`	`int[]`	在 `start` 位置开始的, `len` 个元素的数组的一部分	`subarray('{1,2,3,2,1}'`
`subarray(int[], int start)`	`int[]`	在 `start` 位置开始的数组的一部分	`subarray('{1,2,3,2,1}'`
`intset(int)`	`int[]`	制作单个元素的数组	`intset(42)`

Table F-9. `intarray` 操作符

操作符	返回	描述
int[] && int[]	boolean	重复 — 如果数组至少有一个共同元素则为 true
int[] @> int[]	boolean	包含 — 如果左边的数组包含右边的数组则为 true
int[] <@ int[]	boolean	包含于 — 如果左边的数组包含于右边的数组则为 true
# int[]	int	数组中元素的个数
int[] # int	int	索引 (和 idx 函数相同)
int[] + int	int[]	将元素推入数组中(将元素添加到数组的末尾)
int[] + int[]	int[]	数组串联(右边的数组添加到左边的数组的后面)
int[] - int	int[]	删除匹配右边数组中元素的项
int[] - int[]	int[]	从左边数组中删除右边数组中的元素
int[]	int	int[]	参数的并集
int[]	int[]	int[]	数组的并集
int[] & int[]	int[]	数组的交集
int[] @@ query_int	boolean	如果数组满足查询则为 true (见下文)
query_int ~~ int[]	boolean	如果数组满足查询则为 true (@@ 的交换子)

（PostgreSQL 8.2之前，包含操作符 @> 和 <@ 分别称为 @ 和 ~ 。这些名字仍然可以使用，但是已经废弃了并且最终会被撤销。 请注意，旧的名字从大会移除，之前跟随着核心几何数据类型！）

操作符 && , @> 和 <@ 等同于 PostgreSQL同名的内建操作符，除了它们只工作于整数数组不包含空值， 而内建操作符工作于任意数组类型。这个限制使它们在任何情况下都比内建操作符快的多。

@@ 和 ~~ 操作符测试一个数组是否满足一个*query*， 该查询表示为一个专门的数据类型 query_int 的值。*query* 由针对数组元素检查的整数值组成，可能混合使用操作符 & (与), | (或), 和 ! (非)。在需要时可以使用括号。 例如， 查询 1&(2|3) 匹配包含1也包含2或3的数组。

F.18.2. 索引支持

intarray 为 && , @> , <@ , 和 @@ 操作符还有普通数组相等提供索引支持，

提供两个GiST索引操作符类 ： gist__int_ops （缺省使用） 适合于小到中等的数据集，而 gist__intbig_ops 使用一个大的签名并且更适合于索引大的数据集（也就是， 包含大量不同数组值的字段）。该实现使用一个RD树数据结构和内建的有损压缩。

还有一个非缺省的GIN操作符类 `gin__int_ops` 支持相同的操作符。

GiST和GIN索引的选择取决于GiST和GIN相关的性能特性，这个在别的地方讨论。 一般来说，GIN索引比GiST索引搜索起来更快一些，而建立或更新要慢一些；所以GIN更适合于静态数据，而GiST更适合于经常更新的数据。

F.18.3. 示例

```
-- 一个message可以在一个或多个"sections"中
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- 创建专门的索引
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- 在section 1或2中选择message -OVERLAP操作符
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- 在section 1和2中选择message -CONTAINS操作符
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- 相同的，使用QUERY操作符
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.18.4. 基准

源目录 `contrib/intarray/bench` 包含一个基准测试套件。运行：

```
cd .../bench
createdb TEST
psql TEST < ../_int.sql
./create_test.pl | psql TEST
./bench.pl
```

`bench.pl` 脚本有许多选项，当它不带任何参数的运行时显示。

F.18.5. 作者

所有工作都是Teodor Sigaev (`<[teodor@sigaev.ru](mailto:teodor@sigaev.ru)>`)和 Oleg Bartunov (`<[oleg@sai.msu.su](mailto:oleg@sai.msu.su)>`)完成的。参阅 http://www.sai.msu.su/~megera/postgres/gist/获取额外的信息。 Andrey Oktyabrski在添加新的函数和操作符上做了一个伟大的工作。

F.19. isn

`isn` 模块为下列的国际产品编号标准提供数据类型：EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials)。编号在输入时是经过确认的，根据一个前缀的硬编码列表；这个前缀列表也用于在输出时连接编号。因为不时地分配新的前缀，所以前缀列表可能过期。希望这个模块的将来版本可以根据需要，从一个或多个可以通过用户容易更新的表中获得前缀列表；不过，目前该列表只能通过修改源代码和重新编译来更新。或者，在这个模块的将来版本中，前缀确认或连接支持可能会被删除。

F.19.1. 数据类型

Table F-10显示了 `isn` 模块支持的数据类型。

Table F-10. `isn` 数据类型

数据类型	描述
`EAN13`	European Article Numbers（欧洲文章号），总是以 EAN13 显示格式显示
`ISBN13`	International Standard Book Numbers（国际标准书号），以新EAN13显示格式显示
`ISMN13`	International Standard Music Numbers（国际标准音乐号），以新EAN13显示格式显示
`ISSN13`	International Standard Serial Numbers（国际标准序列号），以新EAN13显示格式显示
`ISBN`	International Standard Book Numbers（国际标准书号），以旧的短显示格式显示
`ISMN`	International Standard Music Numbers（国际标准音乐号），以旧的短显示格式显示
`ISSN`	International Standard Serial Numbers（国际标准序列号），以旧的短显示格式显示
`UPC`	Universal Product Codes（通用产品条码）

一些备注：

1. ISBN13, ISMN13, ISSN13编号都是EAN13编号。
2. EAN13编号并不总是ISBN13, ISMN13 或 ISSN13（有些是）。
3. 一些ISBN13编号可以以ISBN显示。

4. 一些ISMN13编号可以以ISMN显示。
5. 一些ISSN13编号可以以ISSN显示。
6. UPC编号是EAN13编号的一个子集（它们基本上是没有第一个 `0` 数字的EAN13）。
7. 所有UPC, ISBN, ISMN 和 ISSN编号都可以用EAN13编号表示。

在内部，所有这些类型使用相同的表示（一个64位整数），并且所有类型相互之间可以转换。 提供多种类型控制显示格式和允许对应该表示一个编号的特定类型的输入更严格的有效性检查。

`ISBN`, `ISMN`, 和 `ISSN` 类型将在可能时显示编号的短版本(ISxN 10)， 不适合端版本的编号显示ISxN 13格式。 `EAN13`, `ISBN13`, `ISMN13` 和 `ISSN13` 类型总是显示ISxN的长版本(EAN13)。

F.19.2. 转换

`isn` 模块支持下列的类型转换对：

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

当从 `EAN13` 转换到其他类型时，有一个运行时检查，值在其他类型的领域内， 如果不在则抛出一个错误。其他转换只是简单的确认，总是会成功。

F.19.3. 函数和操作符

`isn` 模块提供标准的比较运算符，加上B-tree和哈希索引支持所有的这些数据类型。 另外，有几个专门的函数；显示在Table F-11中。在这个表中， `isn` 意为模块的数据类型之一。

Table F-11. `isn` 函数

函数	返回	描述
`isn_weak(boolean)`	`boolean`	设置weak输入模式 (返回新的设置)
`isn_weak()`	`boolean`	获取当前 weak 模式的状态
`make_valid(isn)`	`isn`	确认一个无效数字(清除无效标识)
`is_valid(isn)`	`boolean`	检查无效标识的存在

*Weak*模式用于可以插入无效数据到表中。无效的意思是检查位为wrong， 而不是说它们是丢失的数字。

为什么想要使用weak模式？很好，可能你有一个巨大的ISBN编号的采集，其中的一些因为怪异的原因有wrong检查位 （可能编号是从打印列表扫描进来的，而OCR（光学字符识别）获取编号错误，也或许编号是手动捕获的......谁知道呢）。 无论如何，关键是你可能想要清理这个烂摊子，但是你又想要在你的数据库中有所有的编号， 或许使用一个额外的工具在数据库中定位无效编号，这样你就可以验证信息并使其更容易的生效； 例如你想要在表中选择所有无效编号。

当你使用weak模式插入无效编号到一个表中时，该编号将带有修正的检查位插入， 但是在显示时将在后面带有一个叹号标记(`!`)，例如 `0-11-000322-5!` 。 这些无效标记可以用 `is_valid` 函数检查然后用 `make_valid` 函数清除。

即使不在weak模式下，你也可以强制插入无效编号，通过在编号后面附加 `!` 字符。

另一个特别特征是在输入期间，可以在检查位上写 `?` ，并且正确的检查位将自动插入。

F.19.4. 例子

```
--直接使用类型：
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--转换类型：
-- 请注意，你只能从ean13转换到另外一个类型当
-- 数字在目标类型的领域中将会是有效的；
-- 因此下列将不会工作: select isbn(ean13('0220356483481'));
-- 但是他们将：
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--创建一个单个字段的表以保持ISBN编码：
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--自动计算校验数位 (观察'?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--使用weak模式：
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.19.5. 参考文献

实施这个模块的信息从几个站点收集而来，包括：

- http://www.isbn-international.org/
- http://www.issn.org/
- http://www.ismn-international.org/
- http://www.wikipedia.org/

用于连字符的前缀也从下列编译而来：

- http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html
- http://www.isbn-international.org/en/identifiers.html
- http://www.ismn-international.org/ranges.html

在算法创建期间要小心，并且他们对在官方 ISBN, ISMN, ISSN 用户手册中建议的算法进行仔细的验证。

F.19.6. 作者

Germán Méndez Bravo (Kronuz), 2004 - 2006

这个模块的灵感来自Garrett A. Wollman的 `isbn_issn` 代码。

F.20. lo

`lo` 模块为管理大对象（也叫做LO或BLOB）提供支持。 包括数据类型 `lo` 和触发器 `lo_manage` 。

F.20.1. 原理

JDBC驱动的问题之一（也影响ODBC驱动），是规范假设BLOB（二进制大对象）的参数是存储在一个表内的， 并且如果该项改变了，那么相关的BLOB会从数据库中删除。

作为PostgreSQL标准，这个不会发生。大对象被视为对象；一个表项可以通过OID引用一个大对象，不过可能多个表项引用同一个大对象OID， 所以系统不会因为你改变或删除一个表项就删除大对象。

这对PostgreSQL专有应用来说很好，但是使用JDBC或ODBC的标准代码不会删除大对象，导致孤独的对象—对象不被任何东西引用，只是简单的占用磁盘空间。

`lo` 模块允许通过在包含LO引用字段的表上附加一个触发器修复这个问题。 该触发器本质上只是在你删除或修改一个引用大对象的值时做 `lo_unlink` 。 当使用这个触发器时，假设只有一个数据库引用被触发器控制字段引用的任意的大对象。

该模块也支持数据类型 `lo` ，该类型实际上只是一个 `oid` 类型的域。 这对于区别持有大对象引用的数据库字段和那些其他事情的OID有帮助。 你不必使用 `lo` 类型来使用该触发器， 但是使用它会很方便的记录你的数据库中的哪个字段代表管理该触发器的大对象， 如果你不为BLOB字段使用 `lo` ，据说ODBC驱动器也会感到迷惑。

F.20.2. 怎样使用

这是一个使用的简单的例子：

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

对于每个将要包含大对象的唯一引用的字段，创建一个 `BEFORE UPDATE OR DELETE` 触发器，并将字段名作为触发器唯一的参数。你也可以通过使用 `BEFORE UPDATE OF` `_column_name_` 限制触发器只在该字段更新时执行。如果你在相同的表中需要多个 `lo` 字段， 那么为每个字段创建一个单独的触发器，记得给相同表上的每个触发器以不同的名字。

F.20.3. 限制

- 删除一个表仍然将孤立它包含的任意对象，因为触发器没有执行。你可以通过在 `DROP TABLE` 之前 `DELETE FROM _table_` 来避免这个问题。

  `TRUNCATE` 有同样的危险。

  如果你已经有或者怀疑你有孤立的大对象，参阅vacuumlo 模块帮助你清除它们。偶尔运行vacuumlo 作为 `lo_manage` 触发器的后备是一个好主意。

- 一些前端可能创建他们自己的表，并且不创建相关的触发器。还有，用户可能不记得（或知道） 创建触发器。

F.20.4. 作者

Peter Mount `<[peter@retep.org.uk](mailto:peter@retep.org.uk)>`

F.21. ltree

这个模块实现了数据类型 `ltree`，该类型表示存储在分层的树形结构里的数据的标签。 提供大量的工具搜索标签树。

F.21.1. 定义

*label*是一个字母数字字符和下划线的序列（例如，在C语言环境， 字符 `A-Za-z0-9_` 是允许的）。标签长度必须少于256字节。

示例：`42`，`Personal_Services`

*label path*是零个或多个由点分隔的标签序列，例如 `L1.L2.L3`， 表示由树状结构的根节点到具体节点的路径。标签路径的长度必须少于65Kb， 但是保持在2Kb以下是最好的。实际上，这不是主要限制；例如， 在DMOZ目录中的最长标签路径(http://www.dmoz.org)大约240字节。

示例：`Top.Countries.Europe.Russia`

`ltree` 模块提供几种数据类型：

- `ltree` 存储标签路径。
- `lquery` 表示一个正规表达式类似的模式以匹配 `ltree` 值。 在一个路径中一个单词匹配那个标签。星号(`*`)匹配零个或多个标签。例如：

  ```
  foo         准确匹配标签路径 `foo`
  *.foo.*     匹配任何包含标签 `foo` 的标签路径
  *.foo       匹配任何以 `foo` 结束的标签路径
  ```

  可以量化星号来限制可以匹配多少个标签：

  ```
  *{n}        准确匹配 n 个标签
  *{n,}       至少匹配 n 个标签
  *{n,m}      至少匹配 n 个但不超过 m 个标签
  *{,m}       最多匹配 m 个标签 — 和  *{0,m 相同}
  ```

  有几个修饰符可以放在 `lquery` 中非星号标签的后面，使其不只是匹配正好的那个：

  ```
  @           匹配大小写无关，例如 a@ 匹配 `A`
  *           匹配任何带有这个前缀的标签，例如 foo* 匹配 foobar
  %           匹配字首下划线分开的单词
  ```

`%` 的行为稍微有点复杂。它试图匹配单词而不是整个标签。例如 `foo_bar%` 匹配 `foo_bar_baz` 而不是 `foo_barbaz` 。如果与 `*` 结合， 前缀匹配分别应用到每个单词，例如 `foo_bar%*` 匹配 `foo1_bar2_baz` 而不是not `foo1_br2_baz` 。

还有，可以写几个由 `|` (OR)分开的标签以匹配任意这些标签， 并且可以在开始放置 `!` (NOT)以匹配任何不匹配任何可选标签的标签。

这是一个带有评注的 `lquery` 的例子：

```
Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
a.  b.     c.      d.               e.
```

这个查询将匹配任何标签路径：

1. 以标签 `Top` 开始
2. 然后有0到2个标签
3. 然后是一个带有大小写不敏感前缀 `sport` 的标签
4. 然后是一个既不匹配 `football` 也不匹配 `tennis` 的标签
5. 然后以一个带有 `Russ` 或准确匹配 `Spain` 开头的标签作为结束。

- `ltxtquery` 表示一个匹配 `ltree` 值的全文本搜索风格模式。 `ltxtquery` 值包含单词，可能在结尾带有修饰词 `@`， `*`， `%`， 这些修饰词的含义和在 `lquery` 中的含义相同。单词可以与 `&` (AND), `|` (OR), `!` (NOT) 和圆括号结合。与 `lquery` 主要的不同是 `ltxtquery` 匹配单词时不考虑单词在标签路径中的位置。

  这是一个 `ltxtquery` 的例子：

  ```
  Europe & Russia*@ & !Transportation
  ```

  这将匹配包含 `Europe` 标签和以 `Russia` （大小写不敏感）开始的标签的路径，但是不匹配包含 `Transportation` 标签的路径。这些单词在路径中的位置并不重要。还有，当使用了 `%` 时， 该单词能匹配标签中任何用下划线分隔单词，不管位置在哪。

注意： `ltxtquery` 允许符号之间有空格，但是 `ltree` 和 `lquery` 不允许这样做。

F.21.2. 操作符和函数

类型 `ltree` 有平常的比较操作符 `=`， `<>`， `<`， `>`， `<=`， `>=`。 比较的优先级以树遍历的顺序，节点的子节点以标签文本排序。另外， 在Table F-12中显示的专业的操作符是可用的。

Table F-12. `ltree` 操作符

操作符	返回	描述
`ltree` `@>` `ltree`	`boolean`	左边参数是右边参数的上代 (或相等)?
`ltree` `<@` `ltree`	`boolean`	左边参数是右边参数的后代 (或相等)?
`ltree` `~` `lquery`	`boolean`	`ltree` 匹配 `lquery` ?
`lquery` `~` `ltree`	`boolean`	`ltree` 匹配 `lquery` ?
`ltree` `?` `lquery[]`	`boolean`	`ltree` 匹配数组中的任意 `lquery` ?
`lquery[]` `?` `ltree`	`boolean`	`ltree` 匹配数组中的任意 `lquery` ?
`ltree` `@` `ltxtquery`	`boolean`	`ltree` 匹配 `ltxtquery` ?
`ltxtquery` `@` `ltree`	`boolean`	`ltree` 匹配 `ltxtquery` ?
`ltree` `||` `ltree`	`ltree`	连接 `ltree` 路径
`ltree` `||` `text`	`ltree`	将文本转化成 `ltree` 并连接
`text` `||` `ltree`	`ltree`	将文本转化成 `ltree` 并连接
`ltree[]` `@>` `ltree`	`boolean`	数组包含 `ltree` 的祖先?
`ltree` `<@` `ltree[]`	`boolean`	数组包含 `ltree` 的祖先?
`ltree[]` `<@` `ltree`	`boolean`	数组包含 `ltree` 的后代?
`ltree` `@>` `ltree[]`	`boolean`	数组包含 `ltree` 的后代?
`ltree[]` `~` `lquery`	`boolean`	数据包含任何匹配 `lquery` 的路径?
`lquery` `~` `ltree[]`	`boolean`	数据包含任何匹配 `lquery` 的路径?
`ltree[]` `?` `lquery[]`	`boolean`	`ltree` 数组包含任意路径匹配任意 `lquery` ?
`lquery[]` `?` `ltree[]`	`boolean`	`ltree` 数组包含任意路径匹配任意 `lquery` ?
`ltree[]` `@` `ltxtquery`	`boolean`	数据包含任意路径匹配 `ltxtquery` ?
`ltxtquery` `@` `ltree[]`	`boolean`	数组包含任意路径匹配 `ltxtquery` ?
`ltree[]` `?@>` `ltree`	`ltree`	第一个数组条目是 `ltree` 的祖先 ?; 如果不是则为 NULL
`ltree[]` `?<@` `ltree`	`ltree`	第一个数组条目是 `ltree` 的后代 ?; 如果不是则为 NULL
`ltree[]` `?~` `lquery`	`ltree`	第一个数组条目匹配 `lquery` ?; 如果不是则为 NULL
`ltree[]` `?@` `ltxtquery`	`ltree`	第一个数组条目匹配 `ltxtquery` ?; 如果不是则为 NULL

操作符 `<@` , `@>` , `@` 和 `~` 类似 `^<@` , `^@>` , `^@` , `^~` ， 除了不用索引之外都相同。这些只是对于测试目的有用。

可用的函数显示在 Table F-13 中。

Table F-13. `ltree` 函数

函数	返回类型	描述	示例
`subltree(ltree, int start, int end)`	`ltree`	`ltree` 的子路径，从位置 `start` 到位置 `end` -1 (从0开始计算)	`subltree('Top.Child1.C`
`subpath(ltree, int offset, int len)`	`ltree`	`ltree` 的子路径，从位置 `offset` 开始，长度为 `len` 。如果 `offset` 为负值，子路径从路径的结尾开始。如果 `len` 是负值，那么距离路径的结尾多少个标签。	`subpath('Top.Child1.Ch`
`subpath(ltree, int offset)`	`ltree`	`ltree` 的子路径，从位置 `offset` 开始，一直到路径的结束。如果 `offset` 为负值，那么子路径从路径的结尾开始。	`subpath('Top.Child1.Ch`
`nlevel(ltree)`	`integer`	路径中的标签的个数	`nlevel('Top.Child1.Chi`

`index(ltree a, ltree b)`	`integer`	`b` 在 `a` 中第一次出现的位置；如果没有发现则为 -1	`index('0.1.2.3.5.4.5.6`
`index(ltree a, ltree b, int offset)`	`integer`	`b` 在 `a` 中第一次出现的位置，从 `offset` 开始搜索；负的 `offset` 意味从路径结尾开始 `-offset` 个标签	`index('0.1.2.3.5.4.5.6`
`text2ltree(text)`	`ltree`	将 `text` 转换为 `ltree`	
`ltree2text(ltree)`	`text`	将 `ltree` 转换为 `text`	
`lca(ltree, ltree, ...)`	`ltree`	最低的公共祖先，也就是，路径的最长公共前缀（支持多达 8 个参数）	`lca('1.2.2.3','1.2.3.4`
`lca(ltree[])`	`ltree`	最低的公共祖先，也就是，路径的最长公共前缀	`lca(array['1.2.2.3'::l`

F.21.3. 索引

`ltree` 支持索引的几个类型，可以加速指出的操作符：

- `ltree` 上的 B-tree 索引： `<` , `<=` , `=` , `>=` , `>`

- `ltree` 上的 GiST 索引：`<`，`<=`，`=`，`>=`，`>`，`@>`，`<@`，`@`，`~`，`?`

  创建这样一个索引的列子：

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

- `ltree[]` 上的 GiST 索引：`ltree[] <@ ltree`，`ltree @> ltree[]`，`@`，`~`，`?`

  创建这样一个索引的例子：

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

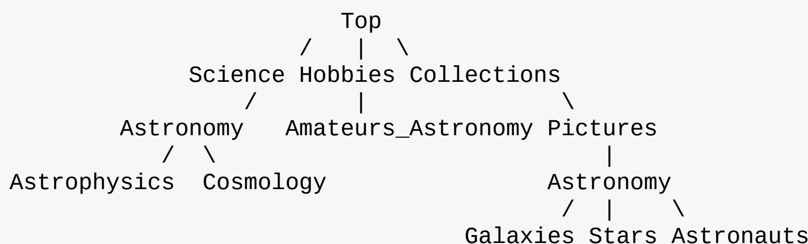
  注意：这种索引类型损耗很大。

F.21.4. 示例

这个示例使用下列的数据（在源代码发布中的`contrib/ltree/ltreetest.sql`文件中也可用）：

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING gist(path);
CREATE INDEX path_idx ON test USING btree(path);
```

现在，我们有一个表`test`，由下面显示的数据描述层级构成：



现在我们可以做继承：

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
------------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

这里是一些路径匹配的例子：

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

这里是一些全文本搜索的例子：

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

路径建造使用函数：

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@ 'T
                 ?column?
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

我们可以通过创建一个SQL函数来简化，该函数在路径中的指定位置插入一个标签：

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@ 'Top.Science.Astrono
                ins_label
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.21.5. 作者

所有工作都是 Teodor Sigaev (`<[teodor@stack.net](mailto:teodor@stack.net)>`) 和 Oleg Bartunov (`<[oleg@sai.msu.su](mailto:oleg@sai.msu.su)>`)做的。参阅 http://www.sai.msu.su/~megera/postgres/gist/获取额外的信息。 作者感谢 Eugeny Rodichev 的有帮助的讨论。欢迎评论和报告bug。

F.22. pageinspect

`pageinspect` 模块提供允许在一个低水平检查数据库页面的内容的函数， 这对于调试目的很有用。所有这些函数只能是超级用户使用。

F.22.1. 函数

`get_raw_page(relname text, fork text, blkno int) returns bytea`

`get_raw_page` 读取命名表指定的块并返回一个拷贝作为 `bytea` 值。 这允许获得一个该块的时间一致的拷贝。 `_fork_` 应该是主数据派生的 `'main'`， 或自由空间映射的 `'fsm'`， 或可见映射的 `'vm'` 。

`get_raw_page(relname text, blkno int) returns bytea`

`get_raw_page` 的短版本，从主分支上读取。 等于 `get_raw_page(relname, 'main', blkno)`

`page_header(page bytea) returns record`

`page_header` 显示了和所有PostgreSQL 堆和索引页相同的字段。

`get_raw_page` 获得的页图像应该作为一个参数传送。例如：

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version | prune_xid
-----------+----------+--------+-------+-------+---------+----------+---------+----------
 0/24A1B50 |        1 |      1 |   232 |   368 |    8192 |     8192 |       4 |         0
```

返回的列符合 `PageHeaderData` 结构中的字段。 参阅 `src/include/storage/bufpage.h` 获取详细信息。

`heap_page_items(page bytea) returns setof record`

`heap_page_items` 显示了一个堆页面中所有的行指针。 对于那些在使用中的行指针，也显示元组头文件。所有的元组都显示， 不管元组在拷贝原始页面时对MVCC快照是否可见。

`get_raw_page` 获得的堆页面图像应该作为一个参数传递。例如：

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

参阅 `src/include/storage/itemid.h` 和 `src/include/access/htup_details.h` 获取返回的字段的解释。

`bt_metap(relname text) returns record`

`bt_metap` 返回关于B-tree索引的元页的信息。例如：

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 1
level     | 0
fastroot  | 1
fastlevel | 0
```

`bt_page_stats(relname text, blkno int) returns record`

`bt_page_stats` 返回关于B-tree索引的单个页面的摘要信息。例如：

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 256
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

`bt_page_items(relname text, blkno int) returns setof record`

`bt_page_items` 返回关于B-tree索引页中的所有条目的详细信息。例如：

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

`fsm_page_contents(page bytea) returns text`

`fsm_page_contents` 显示了FSM页的内部节点结构。输出是多行字符串， 在该页中的二叉树上每个节点一行。这些节点中只有不是零的才输出。号称"next"的指针， 指向下一个从页面中返回的位置的指针，也输出。

参阅 `src/backend/storage/freespace/README` 获取更多关于FSM页面的结构的信息。

F.23. passwordcheck

`passwordcheck` 模块当用户设置CREATE ROLE 或ALTER ROLE时，检查用户的密码。如果认为一个密码太弱， 那么将拒绝该密码并且命令将带有错误终止。

要启用这个模块，在 `postgresql.conf` 中添加 `'$libdir/passwordcheck'` 到shared_preload_libraries， 然后重启服务器。

你可以通过改变源码调整这个模块为你所需的样子。例如，你可以使用 CrackLib 检查密码—这只需要在 `Makefile` 文件中取消两行的注释， 并重新编译该模块。（因为许可证的原因，我们缺省不能包括CrackLib。） 没有CrackLib，该模块为密码强度强制一些简单的规则， 这些规则可以根据你认为合适的去修改或扩展。

> **Caution**
>
> 为了阻止未加密的口令通过网络发送出去、写到服务器日志或被数据库管理员偷走，PostgreSQL允许用户提供预先加密的口令。 许多客户端程序使用这个功能，并在发送到服务器之前加密口令。

这会限制 `passwordcheck` 模块的有用性，因为那种情况下只能尝试猜测口令。 因为这个原因，如果你的安全需求比较高，那么不建议使用 `passwordcheck` 。 使用一个额外的认证方法（如Kerberos）(参阅 Chapter 19) 比依赖于数据库中的密码更安全。

或者，你可以修改 `passwordcheck` 拒绝预先加密的口令， 但是强制用户以明文的方式设置他们的口令有其自身的安全风险。 |

F.24. pg_buffercache

`pg_buffercache` 模块提供实时检查共享缓存内发生了什么的用途。

该模块提供一个C函数 `pg_buffercache_pages` ，该函数返回一个记录集， 加上一个包裹该函数为了方便使用的视图 `pg_buffercache` 。

缺省情况下取消这两种的公共访问，以防隐藏的安全问题。

F.24.1. `pg_buffercache` 视图

被视图暴露的字段的定义显示在Table F-14里。

Table F-14. `pg_buffercache` 字段

名字	类型	引用	描述
`bufferid`	`integer`	ID, 范围为 1.. `shared_buffers`	
`relfilenode`	`oid`	`pg_class.relfilenode`	关系的文件节点号
`reltablespace`	`oid`	`pg_tablespace.oid`	关系的表空间OID
`reldatabase`	`oid`	`pg_database.oid`	关系的数据库OID
`relblocknumber`	`bigint`	关系的页码	
`relforknumber`	`smallint`	关系的分支编号; 参阅 `include/storage/relfilenode.h`	
`isdirty`	`boolean`	页脏了吗？	
`usagecount`	`smallint`	时钟下摆访问计数	

在共享缓存中每个缓冲区都有一行记录。未使用的缓冲区显示为所有字段为空，除了 `bufferid` 。 共享的系统目录显示为属于数据库零。

因为缓存被所有数据库共享，通常有几页的关系不属于当前数据库。 这意味着某些行在 `pg_class` 中没有匹配的连接行， 或者甚至有不正确的连接。如果你尝试连接 `pg_class` ，限制连接到的行的 `reldatabase` 等于当前数据库的OID或0是个好主意。

当访问 `pg_buffercache` 视图时，认为内部缓冲区锁管理器足够长， 能够拷贝视图将显示的所有的缓冲区状态数据。这保证了视图产生一个一致的结果集， 当不再需要阻塞正常的缓冲区活动时。但是，如果频繁的阅读这个视图， 可能会对数据库性能造成一些影响。

F.24.2. 示例输出

```
regression=# SELECT c.relname, count(*) AS buffers
             FROM pg_buffercache b INNER JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             GROUP BY c.relname
             ORDER BY 2 DESC
             LIMIT 10;

             relname             | buffers
---------------------------------+---------
 tenk2                           |     345
 tenk1                           |     141
 pg_proc                         |      46
 pg_class                        |      45
 pg_attribute                    |      43
 pg_class_relname_nsp_index      |      30
 pg_proc_proname_args_nsp_index  |      28
 pg_attribute_relid_attnam_index |      26
 pg_depend                       |      22
 pg_depend_reference_index       |      20
(10 rows)
```

F.24.3. 作者

Mark Kirkwood `<[markir@paradise.net.nz](mailto:markir@paradise.net.nz)>`

设计建议：Neil Conway `<[neilc@samurai.com](mailto:neilc@samurai.com)>`

调试建议：Tom Lane `<[tgl@sss.pgh.pa.us](mailto:tgl@sss.pgh.pa.us)>`

F.25. pgcrypto

`pgcrypto` 模块为PostgreSQL提供cryptographic函数。

F.25.1. 一般散列函数

F.25.1.1. `digest()`

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

计算给定 `data` 的二进制散列。 `type` 是要使用的算法。 标准算法是 `md5`， `sha1`， `sha224`， `sha256`， `sha384` 和 `sha512` 。 如果 `pgcrypto` 带有OpenSSL建立，那么更多算法可用，在 Table F-18中详细说明。

如果你希望digest作为一个十六进制字符串，那么在结果上使用 `encode()` 。 例如：

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. `hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key text, type text) returns bytea
```

为带有键 `key` 的 `data` 计算散列的MAC。 `type` 和在 `digest()` 中相同。

类似于 `digest()` 但是散列只能在知道键的时候计算。 这样就阻止了某个人更改数据并改变匹配的散列的情况。

如果键比散列块大小要大，那么将首先把键散列然后散列的结果作为键使用。

F.25.2. 口令散列函数

函数 `crypt()` 和 `gen_salt()` 是特别为散列口令设计的。 `crypt()` 做散列法， `gen_salt()` 为其准备算法参数。

`crypt()` 中的算法与普通散列算法（如MD5或SHA1）有以下方面的不同：

1. 他们的速度很慢。因为数据很少，所以这是唯一的让蛮力破解口令困难些的方法。

2. 它们使用随机值，称为*salt*，所以有相同口令的用户将会有不同加密了的口令。 也是也对反向算法的附加防御。
3. 它们在结果中包括算法类型，所以不同算法的口令散列可以共存。
4. 它们中的一些是自适应的，这意味着当计算机更快速时，你可以将算法调整的慢一些，而不会引入与现有口令的不相容。

Table F-15列出了 `crypt()` 函数支持的算法。

Table F-15. `crypt()` 支持的算法

算法	最大口令长度	自适应?	Salt位	描述
`bf`	72	yes	128	基于Blowfish，2a的变体
`md5`	unlimited	no	48	基于MD5加密
`xdes`	8	yes	24	扩展的DES
`des`	8	no	12	原始的UNIX加密

F.25.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

计算一个 `password` 的crypt(3)类型散列。当存储一个新的口令时， 需要使用 `gen_salt()` 生成一个新的 `salt` 值。 要检查一个口令，作为 `salt` 传递存储的散列值， 然后检验结果是否匹配存储的值。

设置一个新的口令的示例：

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

认证的示例：

```
SELECT pswhash = crypt('entered password', pswhash) FROM ... ;
```

如果输入的口令是正确的这个就返回 `true` 。

F.25.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

为 `crypt()` 的使用生成一个新的随机salt字符串。 salt字符串也告诉 `crypt()` 使用哪种算法。

`type` 参数指定散列算法。接受的类型有：`des`，`xdes`，`md5` 和 `bf`。

`iter_count` 参数让用户指定重复计数，为这一个算法。计数值越高， 拿它去散列口令的次数越多，因此解开它的次数也越多。尽管太高的计数来计算一个散列可能会用几年的时间， 这有点不切实际。如果省略了 `iter_count` 参数，那么使用缺省的重复计数。 `iter_count` 的允许值取决于算法，在Table F-16中显示。

Table F-16. `crypt()` 的重复计数

算法	缺省	最小	最大
`xdes`	725	1	16777215
`bf`	6	4	31

对于 `xdes`， 这里有一个附加的限制，那就是重复计数必须是奇数。

要选择一个合适的重复计数，考虑原始的DES加密设计是要在那个时间的硬件上每秒有4个散列的速度。 比4个散列每秒慢的可能会降低可用性。高于100散列每秒的可能太快了。

Table F-17给出了不同散列算法的相对缓慢的概述。 该表显示了在8字符口令里尝试所有字符的组合将会花费多长时间，假设口令只包含小写字母， 或者包含大小写字母和数字。 在 `crypt-bf` 记录中， 斜线后的数字是 `gen_salt` 的 `iter_count` 参数。

Table F-17. 散列算法速度

算法	散列/sec	对于 `[a-z]`	对于 `[A-Za-z0-9]`
`crypt-bf/8`	28	246 年	251322 年
`crypt-bf/7`	57	121 年	123457 年
`crypt-bf/6`	112	62 年	62831 年
`crypt-bf/5`	211	33 年	33351 年
`crypt-md5`	2681	2.6 年	2625 年
`crypt-des`	362837	7 天	19 年
`sha1`	590223	4 天	12 年
`md5`	2345086	1 天	3 年

注意：

- 使用的这个机器是1.5GHz Pentium 4。
- `crypt-des` 和 `crypt-md5` 计算的数字是从 John the Ripper v1.6.38 `-test` 的输出获得的。
- `md5` 数字来自mdcrack 1.2。

- `sha1` 数字来自lcrack-20031130-beta。
- `crypt-bf` 数字使用一个简单的程序获得，这个程序重复超过1000次8字符口令。 这样可以显示速度和不同数字的迭代。例如：`john -test` 显示了 `crypt-bf/5` 的213次循环/秒。（结果中非常小的不同与事实一致， `pgcrypto` 中的 `crypt-bf` 实现和John the Ripper中使用的是同一个。）

请注意，"尝试所有组合"是不现实的。不寻常的密码破解在字典的帮助下完成， 包含普通的单词和它们的各种转变。所以，即使有点类似单词的密码可能比上述建议的数字破解的更快， 而一个6字符不像单词的密码可能避开破解。或者不能。

F.25.3. PGP 加密功能

该功能实现了部分OpenPGP (RFC 4880)标准的加密。支持对称秘钥和公共秘钥的加密。

一条加密的PGP消息包含2个部分，或数据包：

- 数据包包含一个会话秘钥—加密了的对称秘钥或者是公共秘钥。
- 数据包包含带有会话秘钥的加密数据。

当带有对称秘钥（如一个口令）加密时：

1. 给定的口令使用String2Key (S2K)算法散列。这和 `crypt()` 算法很相似— 自觉地变慢并且带有随机salt—但是它产生一个全长的二进制秘钥。
2. 如果需要一个单独的会话秘钥，将会产生一个新的随机秘钥。否则将直接使用S2K秘钥作为会话秘钥。
3. 如果直接使用S2K秘钥，那么只有S2K设置将被放入到会话秘钥包。 否则会话秘钥将用S2K秘钥加密然后放入会话秘钥包。

当使用公共秘钥加密时：

1. 将会产生一个新的随机会话秘钥。
2. 它使用公共密钥加密并放入会话秘钥包中。

两种情况下数据被加密的处理如下：

1. 可选的数据操作：压缩，转换成UTF-8，和/或行尾的转换。
2. 数据带有一块随机字节的前缀。这相当于使用一个随机的IV。
3. 附加上一个随机前缀和数据的SHA1散列。
4. 所有这些都带有会话秘钥加密，并放入数据包中。

F.25.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

带有一个对称的PGP秘钥 `psw` 加密 `data` 。 `options` 参数可以包含选项设置，就像下面描述的那样。

F.25.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

解密一个对称秘钥加密的PGP信息。

用 `pgp_sym_decrypt` 解密 `bytea` 数据是不允许的。 这是为了避免输出不合法的字符数据。用 `pgp_sym_decrypt_bytea` 解密原始的文本数据是可以的。

`options` 参数可以包含选项设置，就像下面描述的那样。

F.25.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

用一个公共的PGP秘钥 `key` 加密 `data` 。 给这个函数一个秘密秘钥将产生一个错误。

`options` 参数可以包含选项设置，就像下面描述的那样。

F.25.3.4. `pgp_pub_decrypt()`

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

解密一个公共密钥加密的信息。 `key` 必须是与用来加密的公共秘钥对应的秘密秘钥。 如果该秘密秘钥是密码保护的，你必须在 `psw` 中给出密码。 如果没有密码，但是你希望指定选项，你需要给出一个空的密码。

用 `pgp_pub_decrypt` 解密 `bytea` 数据是不允许的。 这是为了避免输出不合法的字符数据。用 `pgp_pub_decrypt_bytea` 解密原始的文本数据是可以的。

`options` 参数可以包含选项设置，就像下面描述的那样。

F.25.3.5. `pgp_key_id()`

```
pgp_key_id(bytea) returns text
```

`pgp_key_id` 摘取一个PGP公共或秘密秘钥的秘钥 ID。 或如果给出一个加密的信息，它给出用于加密数据的秘钥 ID。

它可以返回两个特殊的秘钥 ID：

- `SYMKEY`

  该信息是用对称秘钥加密的。

- `ANYKEY`

  该信息是公共秘钥加密的，但是秘钥ID已经删除了。这意味着你将要尝试所有你的秘密秘钥， 看看哪个能解密它。 `pgcrypto` 本身并不产生这样的信息。

请注意，不同的秘钥可能有相同的ID。这是稀少的，但是是一个普通事件。 然后客户端应用应该尝试解密每一个，看看哪个合适—类似处理 `ANYKEY` 。

F.25.3.6. `armor()` , `dearmor()`

```
armor(data bytea) returns text
dearmor(data text) returns bytea
```

这些功能打包/解包二进制数据到PGP ASCII-armor格式， 这些基本上是带有CRC的Base64和额外的格式。

F.25.3.7. PGP功能的选项

选项的命名类似于GnuPG。选项的值应该在等号后面给出；选项之间用逗号隔开。例如：

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

除了 `convert-crlf` 之外的所有选项只应用到加密函数。 解密函数从PGP数据中获得参数。

最有趣的选项可能就是 `compress-algo` 和 `unicode-mode` 了。 其余的应该有合理的默认值。

F.25.3.7.1. cipher-algo

要使用的密码算法。

值: bf, aes128, aes192, aes256 (OpenSSL-only: `3des` , `cast5`) 缺省: aes128 适用于: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.7.2. compress-algo

要使用的压缩算法。只有PostgreSQL带有zlib建立时可以使用。

值: 0 - 没有压缩 1 - ZIP 压缩 2 - ZLIB 压缩 (= ZIP 加上元数据和块 CRCs) 缺省: 0 适用于: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.7.3. 压缩级别

压缩多少。较高层次压缩较小但是较慢。0表示禁用压缩。

值: 0, 1-9 缺省: 6 适用于: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.7.4. 转换 crlf

在加密时是否将 `\n` 转换为 `\r\n` 和在解密时是否将 `\r\n` 转换为 `\n`。RFC 4880指定文本数据应该使用 `\r\n` 换行存储。使用这个获得全部的RFC兼容性能。

值: 0, 1 缺省: 0 适用于: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.25.3.7.5. 禁用 mdc

不要用SHA-1保护数据。唯一使用这个选项的理由是为了实现与古老的PGP产品的兼容， 该产品早于SHA-1受保护的包添加到RFC 4880。最近的gnupg.org和pgp.com软件也很好的支持它。

值: 0, 1 缺省: 0 适用于: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.7.6. 启用会话秘钥

使用单独的会话秘钥。公共秘钥加密总是使用一个单独的会话秘钥；这是为了对称秘钥加密， 这在默认情况下是直接使用S2K秘钥的。

值: 0, 1 缺省: 0 适用于: pgp_sym_encrypt

F.25.3.7.7. s2k 模式

使用S2K算法。

值: 0 - 没有salt。 危险的! 1 - 有salt但是带有固定的重复计数。 3 - 变量重复计数。 缺省: 3 适用于: pgp_sym_encrypt

F.25.3.7.8. s2k 摘要算法

在S2K计算中使用哪个摘要算法。

值: md5, sha1 缺省: sha1 适用于: pgp_sym_encrypt

F.25.3.7.9. s2k 密码算法

加密单独的会话秘钥使用哪个密码。

值: bf, aes, aes128, aes192, aes256 缺省: use cipher-algo 适用于: pgp_sym_encrypt

F.25.3.7.10. unicode 模式

是否要转换文本数据从数据库内部编码到UTF-8及以前。如果你的数据库已经是UTF-8， 将不需要转换，但是消息将被标记为UTF-8。没有这个选项将不会这样。

值: 0, 1 缺省: 0 适用于: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8. 用 GnuPG 产生 PGP 秘钥

要生成一个新的秘钥：

```
gpg --gen-key
```

首选的秘钥类型是"DSA and Elgamal"。

对于RSA加密，你必须创建DSA或RSA唯一签署秘钥作为主秘钥，然后用 `gpg --edit-key` 添加一个RSA加密子秘钥。

要列出秘钥：

```
gpg --list-secret-keys
```

以ASCII-armor格式导出一个公共秘钥：

```
gpg -a --export KEYID > public.key
```

以ASCII-armor格式导出一个秘密秘钥：

```
gpg -a --export-secret-keys KEYID > secret.key
```

在将它们送给PGP函数之前需要在这些秘钥上使用 `dearmor()` 。 或者如果你可以处理二进制数据，你可以从命令行中删除 `-a` 。

要获取更多详细信息，请参阅 `man gpg` ， The GNU Privacy Handbook和其他 http://www.gnupg.org上的文档。

F.25.3.9. PGP 代码的限制

- 不支持签名。这也意味着不检查加密子秘钥是否属于主秘钥。
- 不支持加密秘钥作为主秘钥。因为通常不建议这样的做法，这应该不是一个问题。
- 不支持几个子秘钥。这可能看起来像是一个问题，因为这是习惯的做法。另一方面， 不应该使用带有 `pgcrypto` 的定期GPG/PGP秘钥，而是创建一个新的秘钥， 因为使用场景相当不同。

F.25.4. 行加密功能

这些功能在数据上只运行一个密码；它们没有任何比PGP加密更先进的特性。 因此它们有一些主要的问题：

1. 它们使用用户秘钥直接作为加密秘钥。
2. 它们不提供任何完整性检查，来看看加密的数据是否被修改了。
3. 它们希望用户自己管理所有加密参数，即使是IV。
4. 它们不处理文本。

所以，随着PGP加密的引入，不建议使用行加密功能了。

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

加密/解密数据使用 `type` 指定的加密方法。 `type` 字符串的语法是：

```
_algorithm_ [ `-` `_mode_` ] [ `/pad:` `_padding_` ]
```

而 `_algorithm_` 是下列之一：

- `bf` — Blowfish
- `aes` — AES (Rijndael-128)

`_mode_` 是下列之一：

- `cbc` — 下一个块取决于前一个块（缺省）
- `ecb` — 每个块单独加密（只为了测试）

`_padding_` 是下列之一：

- `pkcs` — 数据可以是任意长度（缺省）

- `none` — 数据必须是加密块尺寸的几倍

所以，例如，这些是相等的：

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

在 `encrypt_iv` 和 `decrypt_iv` 中， `iv` 参数是CBC模式的初始值；在ECB中忽略。如果不正好是块的大小则截断或用0补齐。 在没有这个参数的函数里缺省全部为0。

F.25.5. 随机数据函数

```
gen_random_bytes(count integer) returns bytea
```

密码强随机字节的返回 `count` 。一次最多可以提取1024个字节。 这是为了避免排干随机发生器池。

F.25.6. 注意

F.25.6.1. 配置

`pgcrypto` 根据主PostgreSQL `configure` 脚本的调查结果配置它本身。 影响它的选项是 `--with-zlib` 和 `--with-openssl` 。

当用zlib编译时，PGP加密函数可以在加密之前压缩数据。

当用OpenSSL编译时，有更多算法可用。公共秘钥加密函数也会更快， 因为OpenSSL有更多优化了的BIGNUM函数。

Table F-18. 带有和不带有 **OpenSSL** 的功能性总结

功能性	内建	带有 **OpenSSL**
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes (注意 1)
其他摘要算法	no	yes (注意 2)
Blowfish	yes	yes
AES	yes	yes (注意 3)
DES/3DES/CAST5	no	yes
行加密	yes	yes
PGP 对称加密	yes	yes
PGP 公共秘钥加密	yes	yes

注意：

1. SHA2算法在版本 0.9.8 的时候添加到了OpenSSL。对于更老的版本， `pgcrypto` 使用内建的代码。
2. 任何OpenSSL支持的摘要算法是自动获得的。这对于密码来说是不可能的，密码需要明确的支持。
3. AES自版本 0.9.7 以来包含在OpenSSL中了。对于更老的版本， `pgcrypto` 使用内建的代码。

F.25.6.2. NULL 处理

就像SQL中的标准，如果任一参数是NULL，那么所有函数都返回NULL。 这在粗心的使用中可能会造成安全风险。

F.25.6.3. 安全限制

所有 `pgcrypto` 函数在数据库服务器内部运行。这意味着 `pgcrypto` 和客户端应用之间的所有数据和口令移动都是以明文的形式。因此必须：

1. 本地连接或使用SSL连接。
2. 同时信任系统和数据库管理员。

如果你做不到，那么最好在客户端应用内部做crypto。

F.25.6.4. 有用的阅读

* http://www.gnupg.org/gph/en/manual.html

  GNU 隐私手册。

* http://www.openwall.com/crypt/

  crypt-blowfish算法描述。

* http://www.stack.nl/~galactus/remailers/passphrase-faq.html

  如何选择一个好的密码。

* http://world.std.com/~reinhold/diceware.html

  选择密码的有趣想法。

* http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

  描述密码学的优劣。

F.25.6.5. 技术参考文献

* http://www.ietf.org/rfc/rfc4880.txt

  OpenPGP 消息格式。

* http://www.ietf.org/rfc/rfc1321.txt

  MD5 消息摘要算法。

* http://www.ietf.org/rfc/rfc2104.txt

  HMAC:散列的消息认证。

* http://www.usenix.org/events/usenix99/provos.html

  crypt-des、crypt-md5和bcrypt算法的比较。

* http://csrc.nist.gov/cryptval/des.htm

  DES、3DES和AES标准。

* http://en.wikipedia.org/wiki/Fortuna_(PRNG))

  Fortuna CSPRNG的描述。

* http://jlcooke.ca/random/

  基于Jean-Luc Cooke Fortuna的Linux `/dev/random` 驱动器。

* http://research.cyber.ee/~lipmaa/crypto/

密码学指针集合。

F.25.7. 作者

Marko Kreen `<[markokr@gmail.com](mailto:markokr@gmail.com)>`

`pgcrypto` 使用来自下列源码的代码：

算法	作者	起源
DES 加密	David Burren 和其他人	FreeBSD libcrypt
MD5 加密	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish 加密	Solar Designer	www.openwall.com
Blowfish 密码	Simon Tatham	PuTTY
Rijndael 密码	Brian Gladman	OpenBSD sys/crypto
MD5 和 SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

`pg_freespacemap` 模块提供一种检查自由空间映射（FSM）的手段。 它提供一个名为 `pg_freespace` 的函数，或精确的说是两个重载函数。 该函数在一个给定的页面或关系中的所有页面的自由空间映射内显示记录的值。

缺省的公共访问在该函数中取消了，只是因为潜藏的安全问题。

F.26.1. 函数

`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`

返回关系的页面上的自由空间的数量，通过 `blkno` 指定，根据FSM。

`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`

在关系的每个页面上显示自由空间的数量，根据FSM。返回一组 `(blkno bigint, avail int2)` 元组，在关系中每个页面一个元组。

存储在自由空间映射中的数据是不精确的。它们圆整到1/256的 `BLCKSZ` （缺省32字节）的精度， 并且它们不保持全部的元组是最新的（插入了和更新了）。

对于索引，跟踪的是完全未使用的页面，而不是页面中的自由空间。因此， 该值是没有意义的，只表示一个页面是否为空。

> **Note:** 接口在版本8.4中改变了，为了反映在相同版本中引进的新的FSM实现。

F.26.2. 示例输出

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.26.3. 作者

Mark Kirkwood <markir@paradise.net.nz> 制作的原始版本。 在版本8.4中重写，以适应Heikki Linnakangas <heikki@enterprisedb.com> 制作的新的FSM实现。

F.27. pgrowlocks

`pgrowlocks` 模块提供一个显示指定的表的行锁定信息的函数。

F.27.1. 概述

```
pgrowlocks(text) returns setof record
```

该参数是一个表的名字。结果是一组记录，一行代表表中的一个锁定的行。 输出字段在Table F-19中显示。

Table F-19. `pgrowlocks` 输出字段

名字	类型	描述
`locked_row`	`tid`	锁定的行的元组 ID (TID)
`locker`	`xid`	锁定的事务 ID，或如果是多事务则为 multixact ID
`multi`	`boolean`	如果锁定的是多事务则为真
`xids`	`xid[]`	锁定的事务 ID(如果有多个事务则为多个)
`lock_type`	`text[]`	锁定的锁模式 (如果有多个事务则为多个)， 一些 `Key Share` ， `Share` ， `For No Key Update` ， `No Key Update` ， `For Update` ， `Update` 。
`pids`	`integer[]`	锁定后端的过程 ID (如果有多个事务则为多个)

`pgrowlocks` 将 `AccessShareLock` 作为目标表， 并且一行一行的读取行以采集行锁的信息。这对于大表来说不是非常快。请注意；

1. 如果表作为一个整体是被他人排他锁的，那么 `pgrowlocks` 将被阻塞。
2. `pgrowlocks` 不保证生成一个自我一致的快照。 在执行期间，获得一个新的行锁或释放一个旧锁都是可能的。

`pgrowlocks` 并不显示锁定的行的内容。 如果你希望同一时间查看行的内容，你可以像下面这样做：

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

不论如何都要小心(自PostgreSQL 8.3起)，这样一个查询将会是非常低效率的。

F.27.2. 示例输出

```
test=# SELECT * FROM pgrowlocks('t1');
 locked_row | lock_type | locker | multi |   xids    |     pids
------------+-----------+--------+-------+-----------+---------------
      (0,1) | Shared    |     19 | t     | {804,805} | {29066,29068}
      (0,2) | Shared    |     19 | t     | {804,805} | {29066,29068}
      (0,3) | Exclusive |    804 | f     | {804}     | {29066}
      (0,4) | Exclusive |    804 | f     | {804}     | {29066}
(4 rows)
```

F.27.3. 作者

Tatsuo Ishii

F.28. pg_stat_statements

`pg_stat_statements`模块提供一种跟踪执行统计服务器执行的所有SQL语句的手段。

该模块必须通过在`postgresql.conf`中添加`pg_stat_statements`到shared_preload_libraries来加载，因为它需要额外的共享内存。 这意味着添加或删除这个模块都需要重启服务器。

F.28.1. `pg_stat_statements`视图

该模块聚集的统计通过一个名为`pg_stat_statements`的系统视图使其可用。 这个模块为每个不同的查询、数据库ID和用户ID（取决于该模块可以追踪的不同语句的最大值） 包含一行。视图的字段显示在Table F-20中。

Table F-20. `pg_stat_statements` 字段

名字	类型	参考	描述
`userid`	`oid`	`pg_authid`.oid	执行该语句的用户的OID
`dbid`	`oid`	`pg_database` .oid	执行该语句的数据库的OID
`query`	`text`	有代表性的语句的文本 (多达 track_activity_query_size 字节)	
`calls`	`bigint`	执行的次数	
`total_time`	`double precision`	该语句花费的总时间，以毫秒计	
`rows`	`bigint`	该语句恢复或影响的行的总数	
`shared_blks_hit`	`bigint`	该语句命中的共享块缓存的总数	
`shared_blks_read`	`bigint`	该语句读取的共享块的总数	
`shared_blks_dirtied`	`bigint`	该语句弄脏的共享块的总数	
`shared_blks_written`	`bigint`	该语句写入的共享块的总数	
`local_blks_hit`	`bigint`	该语句命中的本地块缓存的总数	
`local_blks_read`	`bigint`	该语句读取的本地块的总数	
`local_blks_dirtied`	`bigint`	该语句弄脏的本地块的总数	
`local_blks_written`	`bigint`	该语句写入的本地块的总数	
`temp_blks_read`	`bigint`	该语句读取的临时块的总数	
`temp_blks_written`	`bigint`	该语句写入的临时块的总数	
`blk_read_time`	`double precision`	该语句读取块花费的总时间，以毫秒计 （如果启用了 track_io_timing，否则为0）	
`blk_write_time`	`double precision`	该语句写入块花费的总时间，以毫秒计 （如果启用了 track_io_timing，否则为0）	

这个视图和函数 `pg_stat_statements_reset` ，只有在通过安装 `pg_stat_statements` 扩展特别安装到的数据库中可用。 不过，当 `pg_stat_statements` 模块加载到服务器中时， 统计跟踪该服务器中的所有数据库，不管该视图是否存在。

为了安全起见，不允许非超级用户查看其它用户执行的查询的文本。不过， 如果视图已经安装到他们的数据库中，那么他们可以看到统计。

可计划的查询（也就是， `SELECT` , `INSERT` , `UPDATE` , 和 `DELETE` ）组合成为一个 `pg_stat_statements` ， 当它们根据一个内部哈希计算有相同的查询结构时。典型的，如果两个查询语义上相等， 除了查询中字面常量的值之外，我们认为这两个查询相同。工具命令（也就是，所有其他命令） 是直接基于它们的文本查询字符串比较的。

当一个常量的值为了匹配其他查询而忽略时，该常量在 `pg_stat_statements` 的显示中被 `?` 替代。查询文本的剩余部分是第一个查询特定散列值与 `pg_stat_statements` 相关条目。

在一些情况下，带有明显不同文本的查询可能合并到一个 `pg_stat_statements` 。 通常这只在语义相等的查询上发生，但是有很小的可能哈希冲突导致不相关的查询被合并到一个条目。（不过，这对于属于不同用户或数据库的查询来说是不会发生的。）

因为哈希值是基于分析查询的表示法之后来计算的，相反的也是可能的： 带有相同文本的查询可能表现为单独的条目，如果它们因为一个因素的结果有不同的含义， 比如不同的 `search_path` 设置。

F.28.2. 函数

`pg_stat_statements_reset() returns void`

`pg_stat_statements_reset` 抛弃所有 `pg_stat_statements` 到目前为止收集的统计。缺省的，这个函数只能被超级用户执行。

F.28.3. 配置参数

`pg_stat_statements.max` (`integer`)

`pg_stat_statements.max` 是该模块追踪语句的最大值 （也就是， `pg_stat_statements` 视图中的最大行数）。 如果观察了比这更多的不同的语句，则会抛弃执行最少的语句的信息。 缺省值是1000。这个参数只能在服务器启动时设置。

`pg_stat_statements.track` (`enum`)

`pg_stat_statements.track` 控制哪个语句可以被该模块计数。 声明 `top` 来跟踪顶级的语句（直接通过客户端发出的语句）。 `all` 也跟踪嵌套的语句（比如包含在函数中的语句）， 或 `none` 禁用语句状态收集。 缺省值是 `top` 。只有超级用户可以更改这个设置。

`pg_stat_statements.track_utility` (`boolean`)

`pg_stat_statements.track_utility` 控制该模块是否追踪工具命令。 工具命令是除了 `SELECT` , `INSERT` , `UPDATE` 和 `DELETE` 的所有命令。缺省值是 `on` 。只有超级用户可以更改这个设置。

`pg_stat_statements.save` (`boolean`)

`pg_stat_statements.save` 指定在服务器关闭时是否保存语句状态。 如果是 `off`， 那么在服务器关闭时不保存状态，在服务器启动时也不重新加载。 缺省值是 `on` 。这个参数只可以在 `postgresql.conf` 文件中或者服务器命令行中设置。

该模块需要额外的共享内存总计大约为 `pg_stat_statements.max` `*` track_activity_query_size字节。请注意， 这个内存在该模块加载时被消耗，即使 `pg_stat_statements.track` 设置为 `none` 。

这些参数必须在 `postgresql.conf` 中设置。典型的用法是：

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.28.4. 示例输出

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]---------------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + ? WHERE bid = ?;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]---------------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + ? WHERE tid = ?;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]---------------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 310.624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]---------------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + ? WHERE aid = ?;
calls       | 3000
total_time  | 271.741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]---------------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 81.42
rows        | 0
hit_percent | 34.4947735191637631
```

F.28.5. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>。 Peter Geoghegan <peter@2ndquadrant.com>添加了查询正常化。

F.29. pgstattuple

`pgstattuple`模块提供各种函数以获得元组级别的状态。

F.29.1. 函数

`pgstattuple(text) returns record`

`pgstattuple`返回一个关系的实际长度，"死的" 元组的百分比，和其他信息。这帮助用户确定是否需要vacuum。 该参数是目标关系的名字（可以有模式限定）。例如：

```
test=&gt; SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

输出字段在Table F-21中描述。

Table F-21. `pgstattuple` 输出字段

字段	类型	描述
`table_len`	`bigint`	实际关系长度以字节计
`tuple_count`	`bigint`	活的元组的数量
`tuple_len`	`bigint`	活的元组的总长度，以字节计
`tuple_percent`	`float8`	活的元组的百分比
`dead_tuple_count`	`bigint`	死的元组的数量
`dead_tuple_len`	`bigint`	死的元组的总长度，以字节计
`dead_tuple_percent`	`float8`	死的元组的百分比
`free_space`	`bigint`	总的空闲空间，以字节计
`free_percent`	`float8`	空闲空间的百分比

`pgstattuple`只获取关系中的读锁。所以结果并不反映一个瞬间的快照；并发的更新将影响它们。

如果`HeapTupleSatisfiesNow`返回假，那么`pgstattuple`判断一个元组是否是"死的"。

`pgstattuple(oid) returns record`

这和 `pgstattuple(text)` 一样，除了目标关系是通过OID指定之外。

`pgstatindex(text) returns record`

`pgstatindex` 返回一个显示关于B-tree索引信息的记录。例如：

```
test=&gt; SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 8192
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 50.27
leaf_fragmentation | 0
```

输出字段是：

字段	类型	描述
`version`	`integer`	B-tree 版本号
`tree_level`	`integer`	根页面的树级别
`index_size`	`bigint`	索引中页面的总数量
`root_block_no`	`bigint`	根块的位置
`internal_pages`	`bigint`	"internal" (上一级) 页面的数量
`leaf_pages`	`bigint`	叶子页面的数量
`empty_pages`	`bigint`	空白页面的数量
`deleted_pages`	`bigint`	已删除页面的数量
`avg_leaf_density`	`float8`	叶子页面的平均密度
`leaf_fragmentation`	`float8`	叶子页面碎片

与 `pgstattuple` 一样，结果是一页一页的累积的， 不应该期望代表整个索引的一个瞬间快照。

`pgstatginindex(regclass) returns record`

`pgstatginindex` 返回一个显示关于GIN索引信息的记录。例如：

```
test=&gt; SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

输出字段是：

字段	类型	描述
`version`	`integer`	GIN 版本号
`pending_pages`	`integer`	等待列表中页面的数量
`pending_tuples`	`bigint`	等待列表中元组的数量

`pg_relpages(text) returns bigint`

`pg_relpages` 返回关系中页面的数量。

F.29.2. 作者

Tatsuo Ishii 和 Satoshi Nagayasu

F.30. pg_trgm

`pg_trgm` 模块提供函数和操作符测定字母数字文本基于三元模型匹配的相似性， 还有支持快速搜索相似字符串的索引操作符类。

F.30.1. 三元模型概念

三元模型是一组从一个字符串中获得的三个连续的字符。 我们可以通过计数两个字符串共享的三元模型的数量来测量它们的相似性。 这个简单的想法证明在测量许多自然语言词汇的相似性时是非常有效的。

> **Note:** `pg_trgm` 从一个字符串提取三元模型时忽略非文字字符（非字母）。 当确定包含在字符串中的三元模型集合时，每个单词被认为有两个空格前缀和一个空格后缀。 例如，字符串"`cat`"中的三元模型的集合是 " `c` "， " `ca` "， " `cat` "和" `at` "。 字符串"`foo|bar`"中的三元模型的集合是 " `f` "， " `fo` "， " `foo` "， " `oo` "， " `b` "， " `ba` "， " `bar` "和" `ar` "。

F.30.2. 函数和操作符

`pg_trgm` 模块提供的函数在Table F-22 中显示，提供的操作符在Table F-23中显示。

Table F-22. `pg_trgm` 函数

函数	返回	描述
`similarity(text, text)`	`real`	返回一个数字表明两个参数是多么相似。结果的范围是0（表明两个字符串完全不相似） 到1（表明两个字符串是完全相同的）。
`show_trgm(text)`	`text[]`	返回一个给定字符串中所有三元模型的数组。（实际上这个除了调试之外很少有用。）
`show_limit()`	`real`	返回 `%` 操作符使用的当前相似性阈值。例如，这个设置两个单词间的最小相似性， 这两个单词被认为足够相似以致相互之间拼写错误。
`set_limit(real)`	`real`	设置 `%` 操作符使用的当前相似性阈值。该阈值必须在0和1之间（缺省是0.3）。 返回传递进来的相同的值。

Table F-23. `pg_trgm` 操作符

操作符	返回	描述
`text` `%` `text`	`boolean`	如果它的参数的相似性高于 `set_limit` 设置的当前相似性阈值则返回 `true` 。
`text <-> text`	`real`	返回参数之间的"距离"，这是1减去 `similarity()` 值。

F.30.3. 索引支持

`pg_trgm` 模块提供GiST和GIN索引操作符类， 该操作符类允许你为了非常快速的相似性搜索在文本字段上创建一个索引。 这些索引类型支持上面描述的相似性操作符，并且额外支持基于三元模型的索引搜索： `LIKE` ， `ILIKE` ， `~` 和 `~*` 查询。 （这些索引并不支持相等也不支持简单的比较操作符，所以你可能也需要一个普通的B-tree索引。）

示例：

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING gist (t gist_trgm_ops);
```

或

```
CREATE INDEX trgm_idx ON test_trgm USING gin (t gin_trgm_ops);
```

在这一点，你将有一个在 `t` 列上的索引，你可以用它来做相似性搜索。 一个典型的查询是：

```
SELECT t, similarity(t, '_word_') AS sml
  FROM test_trgm
  WHERE t % '_word_'
  ORDER BY sml DESC, t;
```

这将返回文本列上与 `_word_` 足够相似的所有值，从最佳匹配到最坏匹配排序。 该索引将用来做一个快速操作，即使是在非常大的数据集上面。

上面查询的一个变体是：

```
SELECT t, t <-> '_word_' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

这个可以通过GiST索引更有效的实现，而不是GIN索引。当只想要少量最靠近的匹配时， 通常会使用第一个公式。

从PostgreSQL 9.1开始，这些索引类型也支持 `LIKE` 和 `ILIKE` 索引搜索，例如

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

该索引搜索通过从搜索字符串中提取三元模型然后在索引中查找这些三元模型来工作。 搜索字符串中的三元模型越多，索引搜索越有效率。不像基于B-tree的搜索， 搜索字符串不需要是左边固定的。

从PostgreSQL 9.3开始，这些索引类型也支持正则表达式匹配的索引搜索 （`~`和`~*`操作符），例如

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

该索引搜索通过从正则表达式中提取三元模型然后在索引中查找这些三元模型来工作。 从正则表达式中提取出来的三元模型越多，索引搜索越有效率。不像基于B-tree的搜索， 搜索字符串不需要是左边固定的。

对于`LIKE`和正则表达式搜索，请记住，没有可提取的三元模型将降级为全文索引搜索。

选择GiST还是GIN索引取决于GiST和GIN的相对性能特征，这个在别的地方讨论。 一般来说，GIN索引比GiST索引搜索起来要快，但是在建立或更新时要慢；索引GIN更适合于静态数据而GiST适合于经常更新的数据。

F.30.4. 文本搜索集成

三元模型匹配在用于与全文本索引相协调时是一个非常有用的工具。 尤其是它可以帮助识别错误拼写的输入单词，这样的单词将不能直接通过全文本搜索机制匹配。

第一步是要生成一个包含文档中的所有唯一词的辅助表：

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

这里的`documents`是含有我们希望搜索的文本字段`bodytext`的表。 使用`simple`配置`to_tsvector`函数的原因， 不是使用一个语言特定的配置，是我们想要一个原始（未修改）词的列表。

下一步，在该词的列上创建一个三元模型索引：

```
CREATE INDEX words_idx ON words USING gin(word gin_trgm_ops);
```

现在，一个类似于之前示例的`SELECT`查询可以用来在用户搜索词中建议错误拼写的词的拼写。 需要一个有用的额外的文本，选中的词和错误拼写的词有相似的长度。

> **Note:** 因为`words`表是作为一个单独的、静态表生成的，它需要定期的重新生成， 这样它才能与文档集合保持合理的更新。通常不需要使它恰好是当前的。

F.30.5. 参考文献

GiST开发网站 http://www.sai.msu.su/~megera/postgres/gist/

Tsearch2开发网站 http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.30.6. 作者

Oleg Bartunov `<[oleg@sai.msu.su](mailto:oleg@sai.msu.su)>`, Moscow, Moscow University, Russia

Teodor Sigaev `<[teodor@sigaev.ru](mailto:teodor@sigaev.ru)>`, Moscow, Delta-Soft Ltd.,Russia

文档：Christopher Kings-Lynne

这个模块是由Delta-Soft Ltd., Moscow, Russia赞助的。

F.31. postgres_fdw

`postgres_fdw` 模块提供外部数据封装器的功能，PostgreSQL通过 它可以访问存储在外部的PostgreSQL服务器上的数据。

本模块提供的功能不但涵盖老版本中dblink模块实现的功能， 而且postgres_fdw提供更加透明和符合标准的语法来访问远程表，并在许多情况下 提供更好的性能。

使用 `postgres_fdw` 模块做远程访问的准备:

1. 使用CREATE EXTENSION语句安装 `postgres_fdw` .
2. 使用CREATE SERVER语句,为每个需要连接的远程数据库 创建一个外部服务器对象。指定除了 `user` 和 `password` 以外的连接信息作为服务器对象的选项。
3. 使用CREATE USER MAPPING语句， 为每个需要通过外部服务器 访问的数据库创建用户映射。指定远程的和密码作为映射用户的 `user` 和 `password` 。
4. 使用CREATE FOREIGN TABLE语句，为每个需要访问的远程表创建外部表。 创建的外部表的对应列必须与远程表匹配。也可以在外部表中使用与远程表不同的表名和列名，但前提是你必须将正确的远程对象名作为创建外部表对象的选项。

上面的操作成功后就可以使用 `SELECT` 外部表的方式访问存储在 远程表中的数据了。同样 `INSERT` ，`UPDATE` 和 `DELETE` 操作都是可以执行的。（映射的远程用户需要有能做这些操作的权限）

建议外部表的字段和相关联的远程表使用相同的数据类型和校对规则，虽然 `postgres_fdw` 允许在需要的时候进行字符类型的转换，但当数据 类型和校对规则不匹配的时候，由于远程服务器和本地服务器对于 `WHERE` 条件的不同解释，也许会造成语义的错误。

需要注意的是一个外部表可以声明比和他关联的远程表更少的列，列的排序也可以 不同。和远程表列的关联用的是列名，和列的位置无关。

F.31.1. postgres_fdw中FDW选项

F.31.1.1. 连接选项

一个作为封装外部数据使用的外部服务器，可以使用libpq接受的连接字符串，详见 Section 31.1.2，除了下面的选项是不允许使用的：

- `user` 和 `password` （将在用户映射中指定）
- `client_encoding` (根据本地服务器编码自动设定）

- `fallback_application_name` (设定为 `postgres_fdw`)

只有超级用户连接到外部服务器是不需要密码的，所以需要为映射的普通用户指定 `password` 。

F.31.1.2. 对象名称选项

这些设置选项被用来控制被发往远程postgres服务器中的sql语句对象名称。 在外部表的名和关联的远程表名不同时，用于设置关联。

`schema_name`

这个选项可以为外部表指定模式名，和远程服务器上的表做关联。如果忽略 这个选项，远程表本身的模式名会被外部表使用。

`table_name`

这个选项可以为外部表指定表名，和远程服务器上的表做关联。如果忽略 这个选项，远程表本身的表名会被外部表使用。

`column_name`

这个选项可以为外部表列指定表名，和远程服务器上的列做关联。如果忽略 这个选项，远程表本身的列名会被外部表使用。

F.31.1.3. 成本估算选项

`postgres_fdw` 检索数据是在远程服务器上执行的，所以成本的估算 不只是远程服务器扫描外部表的效率，还应该加上网络通信的开销。想要获得预期 结果最可靠的方式是对远程服务器做请求增加开销，但是对于一些简单查询来说可能 不值得这样做，所 `postgres_fdw` 提供如下选项做成本估算：

`use_remote_estimate`

外部表或者外部服务器可以指定该选项，用来控制 `postgres_fdw` 是否发出远程的 `EXPLAIN` 命令来获取成本估算。表的设定优先于服务器的设定，但只限于 设定的表。默认值的false。

`fdw_startup_cost`

外部服务器可以指定该选项，该数值类型的选项会在每个外部表的查询开始前加入 一个数值成本。用这个值代表建立连接，在远程端的查询分析和规划的额外开销。 默认值是 `100` 。

`fdw_tuple_cost`

外部服务器可以指定该选项，该数值类型的选项会根据外部表扫描结果为每行加入 一个额外的成本。这个代表服务器间传输数据的额外的网络开销。可以用这个数值 的高低来反应到远程服务器的网络延迟。默认值是 `0.01` 。

`use_remote_estimate` 值为真的时候，成本估算的方法是 `postgres_fdw` 获取远程服务器的语句执行操作成本估算值加上 `fdw_startup_cost` 和 `use_remote_estimate` 。 当值为假的时候，成本估算方法只能是按照语句本地的执行成本加上 `fdw_startup_cost` 和 `use_remote_estimate` 。 除非本地表的统计信息和远程表统计信息的相同，否则本地的估算一般是不精确的。 在外部表执行ANALYZE操作来刷新远程表的统计信息， 这个操作会扫描远程表，使计算和存储统计信息就像在本地一样。在本地保存统计信息可以减少远程表的每个查询的执行计划都造成系统开销。但是如果远程表被更新的频率 太高，本地的统计信息也会很快失去应有的作用。

F.31.1.4. 更新选型

默认情况下，所有 `postgres_fdw` 相关外部表都假设是可以更新的。应用时 优先下面的选项 功能:

`updatable`

这个选项用来控制外部表是否可以用 `INSERT` , `UPDATE` 和 `DELETE` 来修改。该选项可以被外部表或者外部服务器指定。表级别的 选项优先于服务器级别的。默认值是 `true` 。

当然如果一个远程表本身是不能用增删改的，那将会报错。错误直接会在本地抛出。 注意 `information_schema` 将会根据这个选项的设置显示一个外部表是 否可以增删改，而不会去检查远程服务器。

F.31.2. 连接管理

`postgres_fdw` 会在第一个查询外部服务器关联的外部表时 建立和外部服务器的连接。这个连接会一直保持，而且在同一个会话中被重用。 如果涉及多个用户（用户映射）访问外部服务器时，每个用户映射都户建立一个连接。

F.31.3. 事务管理

当查询涉及外部服务器的远程表时，将在本地开启一个对应的事务， `postgres_fdw` 将会在远程服务器也开启事务。 远程事务会和本地事务提交终保持同步。保存点也是一样。

当本地的事务隔离级别为 `SERIALIZABLE` 时，远程的事务隔离级别也使用 `SERIALIZABLE` 。否者远程的隔离级别将是 `REPEATABLE READ` 。 这是为了保证如果一个查询涉及远程服器上多个表的扫描，所有的扫描都会得到一致 的快照结果。这样产生的结果是即使远程服务器上的数据由于其他的操作在更新，单 个事务查询返回的结果也是一样的。上述结果在本地事务隔离级别 `SERIALIZABLE` 和 `REPEATABLE READ` 无论怎样都是可以实现的，但是在 `READ COMMITTED` 下可能会得到意想不到的结果。未来的PostgreSQL发行版本也许会 修改这些规则。

F.31.4. 远程查询优化

postgres_fdw尝试优化远程查询，以减少从外部服务器的数据传输量。通用将带 `WHERE` 查询条件的语句传到远程服务器上执行和不取回与查询结果不相关 的列来实现。为了减少查询未被执行的风险，`WHERE` 从句中都是内建的数据类型 操作和函数的时候才会被传到远程服务器上。在从句中的操作和函数必须是不可变的。

`EXPLAIN VERBOSE` 可以用来检查被送到远程服务器上的查询执行的实际状况。

F.31.5. 版本兼容

`postgres_fdw` 可以将PostgreSQL8.3版本以后的服务器 作为远程服务器使用。8.1,8,2的服务器只能提供读的功能。有这样个问题由于版本的差异 将where中的内建函数送到远程服务器上执行的时候，远程服务器由于版本低无法识别会报 "function does not exist"或者相似的错误。可以用从写sql的方式解决，我们嵌入 一个子查询sub-`SELECT` with `OFFSET 0`,将有问题的函数和操作移 出 sub-`SELECT` 。

F.31.6. 作者

Shigeru Hanada `<[shigeru.hanada@gmail.com](mailto:shigeru.hanada@gmail.com)>`

F.32. seg

这个模块为定义线段和浮点类型的时间间隔提供了一种数据类型 `seg` ， 这种数据类型由于可以代表不确定终点的区间，所以在做实验测量的时候特别有用。

F.32.1. <--!Rationale-->原理

几何学测量要比测量连续的点复杂的多，测量出的结果通常是一些模糊限制的连续值 组成的区间值。这种区间的产生可能是由于测量出结果的不确定性，随机性，也可能 要测量的值本身就是以区间的形式作为条件的，如测量蛋白质稳定性的温度范围。

一般来说，用这个数据类型存储区间值比用双引号更方便。实际上，在大部分的应用 中这么使用也更有效率。

进一步说，传统数值类型存储数据可能对一些模糊限制的值产生偏差，你获取的会 值是6.50并且存入数据库，当你读出的时候会是什么，看下面例子：

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

在精确测量中，6.50和6.5代表的意思是不一样的，而且有时差异会很大。测量者 记下或者公布6.50代表了一个更大甚至更模糊的区间值，6.5只是代表了6.50的一 个中心点。像这种不同数据的同化表达是我们绝对不想看到的。

本章的特殊数据类型可以用来记录任意可变精度的间隔值，而且记录的每个数据都可以 使用自己的精度。

查看结果：

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.32.2. 语法

使用一个或者二个精度数代表一个区间,连接格式（ `..` 或 `...` ）。另外也可以用值和偏移符号表示（ `<` ， `>` 或 `~` ）。（正确的符号会被所有内建 的操作忽略。）表Table F-24和Table F-25 给出了区间表示的方法和一些例子。

在表Table F-24中 _x_ , _y_ 和 _delta_ 代表双精度数。可以在 _x_ , _y_ 之前 添加正确的操作符。

Table F-24. seg External Representations

_x_	单个值（间隔为0）
_x_ .. _y_	区间 _x_ 到 _y_
_x_ (+-) _delta_	区间 _x_ - _delta_ to _x_ + _delta_
_x_ ..	_x_ 开区间无下限
.. _x_	_x_ 开区间无上限

Table F-25. Examples of Valid seg Input

5.0	零长度的区间（一个点）
~5.0	~ 作为一个标记存在，5.0的点和记录
<5.0	< 作为一保留个标记存在，比点5.0小的
>5.0	> 作为一个保留标记存在，比点5.0大的
5(+-)0.3	同区间 4.7 .. 5.3 ， (+-) 不是保留标记
50 ..	区间大于50
.. 0	区间小于0
1.5e-2 .. 2E-2	表示区间 0.015 .. 0.02
1 ... 2	1...2 , 1 .. 2 , 1..2 表达区间相同（空格会被忽略）

在数据源中 ... 被广泛使用，他的作用和另外一种拼写 .. 相同。但这可能产生歧义如 0...23 是表示 23 还是 0.23 . 所以在 seg 数据类型中所有十进制小数的小数点前都需要有一个数字。

seg 不允许表示的区间由大到小表示，如 5 .. 2 。

F.32.3. 精度

seg 存储的值使用32-位的浮点数。精度不超过7位。

精度小于7位，保留本身的精度。换句话说如果你的返回值是0.00，那么这是数值本身带的精度 而不是格式化处理的。数值前的0不作为精度:值0.0067的精度被视为2。

F.32.4. 使用

seg 模块包含可以在 GiST索引中操作 seg 数据类型值的类，该类支持表 Table F-26中的操作。

Table F-26. Seg GiST Operators

操作	描述
[a, b] << [c, d]	[a,b]的范围完全在[c,d]的左侧。换句话说，b < c那么 [a,b] << [c, d] 为真，否则为假。
[a, b] >> [c, d]	[a,b]的范围全部在[c,d]的右侧。换句话说，a > d那么 [a,b] >> [c, d] 为真，否则为假。
[a, b] &< [c, d]	左包含，或者解读为取值范围不超过右边，b <= d时为真。
[a, b] &> [c, d]	右包含，或者解读为取值范围不超过左边，a >= c时为真。
[a, b] = [c, d]	[a, b]和[c, d]相同，当a=c并且b=d时为真。
[a, b] && [c, d]	[a, b]和[c, d]重叠部分
[a, b] @> [c, d]	[a, b]包含[c, d]，意味着 a <= c并且 b >= d。
[a, b] <@ [c, d]	[a, b]被[c, d]包含，意味着 a >= c and b <= d。

PostgresSQL 8.2之前的版本中， @> 和 <@ 分别用 @ 和 ~ 表示。 但那种表示方式会在将来完全被替代，所以不赞成使用。注意旧的名称会随着代表几何的数据类型转换为惯例的格式。

提供标准的B-tree操作，例如

操作	描述
[a, b] < [c, d]	小于
[a, b] > [c, d]	大于

这些操作除了做排序以外没有别的用处，首先是(a)和(c)比较大小，如果 相等在比较(b)和(d)。如果使用这种类型排序，得到的的结果在大多数情 况下是合理的。

F.32.5. 注意

使用例子，参看回归测试 sql/seg.sql 。

使用 (+-) 机制转换数字的时候可能得到不确定的精度。举个例子他可能 为较低的边界值增加额外的精度：

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- 应该为: 9 .. 11
```

R-tree索引的性能很大程度上依赖初始输入值的排序。在输入表的 `seg` 类型列 上排序可能是非常有效果的；参考脚本 `sort-segments.pl` 。

F.32.6. 感谢

原作者：Gene Selkov, Jr. `<[selkovjr@mcs.anl.gov](mailto:selkovjr@mcs.anl.gov)>` ，数学与计算机科学部，阿贡国家实验室。

感谢Joe Hellerstein教授 (http://db.cs.berkeley.edu/jmh/)阐明的思想 (http://gist.cs.berkeley.edu/)。感谢所有 Postgres开发者，使我可以站在巨人的肩膀上。 感谢阿贡实验室和美国能源部多年的忠实支持我的数据库的研究。

F.33. sepgsql

`sepgsql` 是一个可加载的模块，支持基于标签的强制访问控制(MAC)， 以SELinux安全策略为基础。

> **Warning**
>
> 当前的实现有很大的局限性，并不为所有的动作都执行强制访问控制。 参阅Section F.33.7。

F.33.1. 概述

这个模块与SELinux结合，提供一个PostgreSQL 正常提供的安全检查的附加层。从SELinux来看，这个模块允许 PostgreSQL起到用户空间对象管理者的作用。 DML查询发起的每个表和函数访问都将针对系统安全策略做检查。 这个检查是PostgreSQL执行的通常的SQL权限检查之外的东西。

SELinux访问控制决策利用安全标签， 以字符串表示，如 `system_u:object_r:sepgsql_table_t:s0` 。 每个访问控制决策包含两个标签：尝试执行动作的主题的标签， 和要被执行操作的对象的标签。因为这些标签可以用于任意类型的对象， 所以存储在数据库中的对象的访问控制决策会（用这个模块是这样的） 和任意其他类型的对象（比如，文件）一样使用通用标准。 这个设计是为了允许集中的安全策略保护信息资产独立于这些资产的存储细节。

SECURITY LABEL语句允许分配安全标签给数据库对象。

F.33.2. 安装

`sepgsql` 只能用于Linux 2.6.28 或更高的启用SELinux的系统。 在其他的平台上不能使用。也需要libselinux 2.1.10 或更高版本和selinux-policy 3.9.13或更高版本 （尽管一些分支可以移植需要的规则到老的政策版本）。

`sestatus` 命令允许检查SELinux的状态。 一个典型的显示是：

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

如果禁用了SELinux或者没有安装SELinux， 那么必须在安装这个模块之前先配置该产品。

要建立这个模块，在你的PostgreSQL `configure` 命令中包含选项 `--with-selinux`。确保在建立时已经安装了 `libselinux-devel` RPM。

要使用这个模块，必须在 `postgresql.conf` 的 shared_preload_libraries参数中包含了 `sepgsql`。 如果以任何其他方式加载，该模块将不会正确的运行。一旦加载了该模块， 你应该在每个数据库中执行 `sepgsql.sql`。 这将安装安全标签管理需要的函数，并且分配最初的安全标签。

这里是一个示例，显示如何用 `sepgsql` 函数和安装的安全标签初始化新的数据库集群。 为你的安装适当的调整显示的路径：

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  change
    #shared_preload_libraries = ''                # (change requires restart)
  to
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

请注意，你可能会看到一些或者所有下列的通知，取决于你的 libselinux和selinux-policy版本：

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type db_blob
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type db_lang
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type db_lang
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type db_lang
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type db_lang
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type db_lang
```

这些信息是无害的，应该忽略。

如果安装进程没有错误的完成了，那么你现在可以正常的启动服务器。

F.33.3. 回归测试

由于SELinux的性质，为 `sepgsql` 运行回归测试需要几个额外的配置步骤，其中一些必须作为root用户完成。 回归测试不通过一个普通的 `make check` 或 `make installcheck` 命令来运行；你必须设置配置，然后手动的调用测试脚本。 测试必须在配置的PostgreSQL构造树的 `contrib/sepgsql` 目录中运行。 尽管他们需要一个构造树，但是测试的目的是针对一个已经安装的服务器执行的， 这点他们可以与 `make installcheck` 作比较。

第一步，根据Section F.33.2中的指示， 在一个运行的数据库中设置 `sepgsql`。 请注意，当前的操作系统用户必须能够作为超级用户不需要密码认证的连接到数据库。

第二步，为回归测试建立和安装策略包。 `sepgsql-regtest` 策略是一个特定用途的策略包， 提供了一组在回归测试期间被允许的规则。应该从策略源文件 `sepgsql-regtest.te` 中建立，使用 `make` 和SELinux提供的Makefile来实现。 你需要在你的系统上定位适当的Makefile；下面显示的路径只是一个示例。 一旦建立，使用 `semodule` 命令安装这个策略包， 它加载提供的策略包到内核中。如果策略包正确的安装了， 那么 `semodule` -l应该会将 `sepgsql-regtest` 作为一个可用的策略包列出：

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

第三步，打开 `sepgsql_regression_test_mode` 。 出于安全考虑， `sepgsql-regtest` 中的规则缺省是不启用的； `sepgsql_regression_test_mode` 参数启用需要发出回归测试的规则。 可以使用 `setsebool` 命令打开：

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

第四步，验证你的shell是在 `unconfined_t` 域中运行：

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

如果需要，请参阅Section F.33.8获取调整你的工作域的详细信息。

最后，运行回归测试脚本：

```
$ ./test_sepgsql
```

这个脚本将尝试验证你已经正确的做了所有的配置步骤， 然后它将为 `sepgsql` 模块运行归回测试。

完成测试之后，建议你禁用 `sepgsql_regression_test_mode` 参数：

```
$ sudo setsebool sepgsql_regression_test_mode off
```

你可能想要彻底删除 `sepgsql-regtest` 策略：

```
$ sudo semodule -r sepgsql-regtest
```

F.33.4. GUC 参数

`sepgsql.permissive` (`boolean`)

这个参数使得 `sepgsql` 能够在许可的模式运行，无视系统的设置。 缺省为off。这个参数只能在 `postgresql.conf` 文件中或者在服务器的命令行设置。

当这个参数为on时， `sepgsql` 函数在许可模式， 即使SELinux通常运行在强制模式。这个参数对于测试目的尤其有用。

`sepgsql.debug_audit` (`boolean`)

这个参数启用审计信息的打印，无视系统策略设置。缺省是off， 意味着信息将根据系统设置来打印。

SELinux的安全策略也有规则控制是否记录特殊的访问。 缺省的，只记录违规的访问。

这个参数强制打开所有可能的记录，无视系统的策略。

F.33.5. 特性

F.33.5.1. 控制对象类

SELinux的安全模型描述了所有的访问控制规则， 作为主题的实体（通常，数据库的一个客户端）和对象的实体（比如一个数据库对象） 的关系，每一个都由安全标签来鉴别。如果尝试访问一个没有标签的对象， 那么该对象被当做是分配了 `unlabeled_t` 标签。

目前， `sepgsql` 允许分配安全标签给模式、表、字段、序列、 视图和函数。当正在使用 `sepgsql` 时， 自动分配安全标签给在创建时支持的数据库对象。这个标签被称为缺省的安全标签， 并且是根据系统的安全策略决定的，数据库会拿这个系统的安全策略输入来当作创建人的标签， 该标签分配给新对象的父对象，并且可以选择以构造对象名来命名。

一个新的数据库对象基本上继承父对象的安全标签，除了安全策略有特殊的类型转换规则时， 会应用一个不同的标签。对于模式来说，父对象是当前数据库；对于表、序列、 视图和函数， 是包含的模式；对于字段，是包含的表。

F.33.5.2. DML 权限

对于表， `db_table:select` 、 `db_table:insert` 、 `db_table:update` 或 `db_table:delete` ，根据语句的类型， 为所有引用的目标表做检查；另外， `db_table:select` 也为所有在 `WHERE` 或 `RETURNING` 子句中的包含字段引用的表做检查， `UPDATE` 的数据源也是如此，等等。

也将为每个引用的字段检查字段级别的权限。 `db_column:select` 不只是检查被 `SELECT` 读取的字段，也检查在其他DML语句中引用的字段； `db_column:update` 或 `db_column:insert` 也将检查被 `UPDATE` 或 `INSERT` 修改的字段。

例如，考虑：

```
UPDATE t1 SET x = 2, y = md5sum(y) WHERE z = 100;
```

这里，`db_column:update` 将为 `t1.x` 做检查， 因为它被更新了，`db_column:{select update}` 将为 `t1.y` 做检查，因为它被更新和引用了， `db_column:select` 将为 `t1.z` 做检查， 因为它被引用了。 `db_table:{select update}` 也将在表级别做检查。

对于序列，当我们使用 `SELECT` 引用一个序列对象时， 对 `db_sequence:get_value` 做检查；不过， 请注意， 我们当前不检查执行对应的函数的权限，如 `lastval()` 。

对于视图，将检查 `db_view:expand` ， 然后任何其他需要的权限都将分别在从视图扩展的对象上做检查。

对于函数，当用户尝试将函数作为查询的一部分执行，或使用快速路径调用时， 会对 `db_procedure:{execute}` 做检查。如果这个函数是一个受信任的程序， 那么也会检查 `db_procedure:{entrypoint}` 的权限， 看看它是否可以作为受信任的程序的入口点来执行。

为了访问任意模式对象，在包含的模式上需要 `db_schema:search` 权限。 当不带有模式限定的引用一个对象时，这个模式的权限没有出现将不会被搜索 （就好像用户在这个模式上没有 `USAGE` 权限）。 如果给出了明确的模式限定，如果用户在命名的模式上没有必需的权限，那么将会出现一个错误。

客户端必须被允许访问所有引用的表和字段，即使它们起源于随后扩张的视图， 所以我们应用一致的访问控制规则，独立于引用表内容的方式。

缺省的数据库权限系统允许数据库超级用户使用DML命令修改系统目录， 引用和修改toast表。当启用 `sepgsql` 时，禁止这些操作。

F.33.5.3. DDL 权限

SELinux为每个对象类型定义了几个控制一般操作的权限；比如创建、 修改、删除和重新贴安全标签。另外，几个对象类型有特殊的权限控制它们的典型操作； 如在一个特别的模式中添加或删除名字入口。

创建一个新的数据库对象需要 `create` 权限。SELinux 将根据客户端的安全标签授予或拒绝这个权限，并且为新的对象拟建安全标签。 在某些情况下，需要额外的权限：

- CREATE DATABASE还需要源或模板数据库的 `getattr` 权限。
- 创建一个模式对象还需要在父模式上有 `add_name` 权限。
- 创建一个表还需要有权限创建每个单独的表字段，就好像每个表字段是一个独立的顶级对象。

- 创建一个标记为 `LEAKPROOF` 的函数还需要 `install` 权限。（当为一个现有的函数设置 `LEAKPROOF` 时，也需要检查这个权限。）

当执行 `DROP` 命令时，将在要被删除的对象上检查 `drop` 。也要检查通过 `CASCADE` 直接删除的对象的权限。包含在特定模式（表、视图、序列和程序）中对象的删除还需要在该模式上的 `remove_name` 。

当执行 `ALTER` 命令时，将为每个对象类型的被修改的对象检查 `setattr` ，除了附属的对象，如表的索引或触发器，这里的权限是在父对象上检查的。在某些情况下，需要额外的权限：

- 移动一个对象到新的模式还需要在旧的模式上有 `remove_name` 权限，和在新的模式上有 `add_name` 权限。
- 在一个函数上设置 `LEAKPROOF` 属性需要 `install` 权限。
- 在一个对象上使用SECURITY LABEL还需要在与老的安全标签结合的对象上有 `relabelfrom` 权限，和在与新的安全标签结合的对象上有 `relabelto` 权限。（在安装了多个标签提供者和用户尝试设置一个安全标签，但不被SELinux 管理的情况下，只应该检查 `setattr` 。这是由于实现的限制，目前没有做到的。）

F.33.5.4. 受信任的程序

受信任的程序类似于安全定义函数或setuid命令。SELinux 提供一个特性，允许受信任的代码使用一个不同于客户端的安全标签运行，通常是为了提供到敏感数据的高度控制的访问（例如，可能会忽略行，或者存储值的精度可能会减少）。一个函数是否作为受信任的程序动作，是由它的安全标签和操作系统安全策略控制的。例如：

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                        FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

以上的操作应该由管理员用户执行。

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

在这种情况下，普通用户不能直接引用 `customer.credit` ， 但是一个受信任的程序 `show_credit` 允许他带有一些数字标记的打印客户的信用卡号码。

F.33.5.5. 动态域转换

使用SELinux的动态域转换特性来转换客户端程序、客户端域的安全标签到一个新的内容是可能的， 如果安全策略允许这么做。客户端域需要 `setcurrent` 权限， 还有从老的都新的域的 `dyntransition` 权限。

动态域转换应该仔细考虑，因为他们允许用户转换他们的标签， 并且因此他们在选项上的权限不受系统的授权（在受信任的程序的情况下）。 因此， `dyntransition` 权限只在用于转换一个域到更小的权限集合时认为是安全的。 例如：

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-------------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c1023
ERROR:  SELinux: security policy violation
```

在上面的例子中，允许我们从较大的MCS范围 `c1.c1023` 转换到较小的范围 `c1.c4` ，但是反过来转换就被拒绝了。

动态域转换和受信任的程序的组合使得一个有趣的用例满足连接池软件的典型的生命周期过程。 尽管你的连接池软件不能运行大多数的SQL命令，但是你可以允许它使用 `sepgsql_setcon()` 函数从一个受信任的程序里面切换客户端的安全标签；可能需要一些证书批准切换客户端标签的请求。 之后，这个会话将会拥有目标用户的权限，而不是连接池的权限。该连接池可以稍后使用带有 `NULL` 参数的 `sepgsql_setcon()` 函数恢复安全标签的改变， 再次从受信任的程序里面用适当的权限检查调用。 这里的要点是只有那个受信任的程序实际上拥有改变有效的安全标签的权限，并且给出适当的证书。 当然，对于安全的操作，证书的存储（表、过程定义或其他东西）必须防止越权访问。

F.33.5.6. 其他

我们拒绝LOAD命令包括一切， 因为加载的任何模块都能很容易的绕开安全策略实施。

F.33.6. Sepgsql 函数

Table F-27显示了可用的函数。

Table F-27. Sepgsql 函数

| `sepgsql_getcon() returns text` | 返回客户端的域，客户端当前的安全标签。 | | `sepgsql_setcon(text) returns bool` | 如果安全策略允许，则切换当前会话的客户端的域到一个新的域。 也接受 `NULL` 输入作为转换到客户端原来的域的一个请求。 | | `sepgsql_mcstrans_in(text) returns text` | 如果mcstrans进程正在运行，则转化给出的限定MLS/MCS范围为行格式。 | | `sepgsql_mcstrans_out(text) returns text` | 如果mcstrans进程正在运行，则转化给出的行MCS/MCS范围为限定的格式。 | | `sepgsql_restorecon(text) returns bool` | 为当前数据库中所有的对象设置初始的安全标签。 参数可以是NULL， 或特定文件的名字用于系统默认的选择。 |

F.33.7. 限制

数据定义语言 (DDL) 权限

由于实现的限制，一些DDL操作不检查权限。

数据控制语言 (DCL) 权限

由于实现的限制，DCL操作不检查权限。

行级别的访问控制

PostgreSQL不支持行级别的访问；因此 `sepgsql` 也不支持。

隐藏通道

`sepgsql` 没有试图隐藏某一对象的存在，即使不允许用户访问它。 例如，我们可以推断一个不可见对象的存在，根据主键冲突、外键冲突等， 虽然我们不能获得该对象的内容。最高机密表的存在不能被隐藏；我们只希望能隐藏它的内容。

F.33.8. 外部资源

SE-PostgreSQL 介绍

这个wiki页提供了一个简要概述、安全设计、构造、管理和即将到来的特性。

Fedora SELinux 用户指南

这个文档提供了在你的系统上管理SELinux的广泛的知识。 它主要集中于Fedora，但是不局限于Fedora。

Fedora SELinux FAQ

这个文档回答了关于SELinux的常见问题。 它主要集中于Fedora，但是不局限于Fedora。

F.33.9. 作者

KaiGai Kohei `<[kaigai@ak.jp.nec.com](mailto:kaigai@ak.jp.nec.com)>`

F.34. spi

spi模块提供几个使用SPI和触发器的可行的示例。 当这些函数是它们自己正确的某些值时，它们对于你自己的目的是更有用的修改的例子。 该函数一般足够任意的表使用，但是你在创建一个触发器时必须指定表和字段名（正如下面描述）。

下面描述的每一组函数都是作为一个独立可安装的扩展提供的。

F.34.1. refint — 实现参照完整性的函数

`check_primary_key()` 和 `check_foreign_key()` 用来检查外键约束。（这个功能早已被内置的外键机制取代，但是该模块作为一个例子仍然是有用的。）

`check_primary_key()` 检查引用表。为了使用该函数，创建一个 `BEFORE INSERT OR UPDATE` 触发器，该触发器在一个表上使用这个函数引用另一个表。 作为触发器参数指定：来自外键的引用表的字段名，被引用的表名，来自主/唯一键的被引用表的字段名。 要处理多个外键，为每个引用创建一个外键。

`check_foreign_key()` 检查被引用的表。为了使用该函数，创建一个 `BEFORE DELETE OR UPDATE` 触发器， 该触发器在一个表上使用这个函数被另外一个表引用。作为触发器参数指定： 该函数必须执行检查的引用表的数量，如果发现一个引用键的动作（ `cascade` — 删除引用行， `restrict` — 如果引用键退出则退出事务， `setnull` — 设置引用键字段为空）， 来自主/唯一键的被触发表的字段名，然后是引用表名和字段名 （重复引用表的次数和第一个参数指定的一样多）。请注意， 主/唯一键字段应该标记为NOT NULL并且应该有一个唯一索引。

示例在 `refint.example` 。

F.34.2. timetravel — 实现时间行程的函数

很久以前，PostgreSQL有一个内置的时间行程特性， 保持为每个元组插入和删除时间。这个特性可以使用这些函数模仿。 要使用这些函数，必须添加两个 `abstime` 类型的字段到一个表， 用来存储一个元组插入(start_date)和更改/删除(stop_date)的日期：

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

该字段可以随你喜欢任意命名，但是在这个讨论中我们将它们称作start_date和stop_date。

当插入一个新行时，start_date通常设置为当前时间，stop_date设置为 `infinity` 。 如果插入的数据在这些字段中包含空，那么触发器将自动的替换这些值。 通常只应该在重新加载转储的数据时在这些字段中明确的插入非空数据。

stop_date等于 `infinity` 的元组是"现在有效的"，可以修改。 带有限定的stop_date的元组不能再被修改—触发器将阻止修改。 （如果需要修改，可以像下面显示的那样关闭时间行程。）

对于可修改的行，在更新时只有被更新的元组内的stop_date被更改（为当前时间） 并且插入一个带有修改数据的新的元组。在这个新元组内的Start_date设置为当前时间， stop_date设置为 `infinity` 。

删除并不实际删除元组，只是设置它的stop_date为当前时间。

要查询元组的"现在有效"，在查询的WHERE条件中包括 `stop_date = 'infinity'` 。（你可能希望体现到一个视图中。） 相似的，你可以用合适的start_date和stop_date条件查询任意过去时间的元组有效性。

`timetravel()` 是支持这个行为的常规触发器函数。 在每个时间行程表上创建一个使用这个函数的 `BEFORE INSERT OR UPDATE OR DELETE` 触发器。 指定两个触发器参数：start_date和stop_date字段的实际名字。 可选的，你可以再指定一到三个参数，这些参数必须引用类型为 `text` 的字段。 触发器将存储当前用户名到这些字段中，在INSERT期间存储到第一个中，在UPDATE期间存储到第二个中，在DELETE期间存储到第三个中。

`set_timetravel()` 允许为一个表打开或关闭时间行程。 `set_timetravel('mytab', 1)` 将为表 `mytab` 返回TT ON。 `set_timetravel('mytab', 0)` 将为表 `mytab` 返回TT OFF。两种情况下都报道老的状态。当TT为off是，可以自由修改start_date和stop_date字段。 请注意，on/off状态对于当前数据库会话来说是局部的—新的会话将对于所有表来说总是以TT ON开始。

`get_timetravel()` 为一个表返回TT的状态而不会改变这个表。

在 `timetravel.example` 中有一个示例。

F.34.3. autoinc — 自增字段函数

`autoinc()` 是一个存储序列的下一个值到一个整数字段的触发器。 与内置的"序列字段"特性有些重叠，但是并不相同： `autoinc()` 在插入时重写替代一个不同的字段值的尝试，并且可选择的， 它也可以用于在更新时增加字段。

要使用该函数，创建一个使用该函数的 `BEFORE INSERT` (或者可选择的 `BEFORE INSERT OR UPDATE`) 触发器。 指定两个触发器参数：要被修改的整数字段的名字，和将要填充值的序列对象名。 （实际上，可以指定任意数量的这样的名字对，如果想要更新多个自增字段。）

在 `autoinc.example` 中有一个示例。

F.34.4. insert_username — 追踪谁改变了表的函数

`insert_username()`是一个存储当前用户名到一个文本字段的触发器。 这对于追踪谁最后修改了表中指定的行是有用的。

要使用该函数，创建一个使用该函数的`BEFORE INSERT`和/或`UPDATE`触发器。 指定一个触发器参数：要修改的文本字段名。

在`insert_username.example`中有一个示例。

F.34.5. moddatetime — 追踪最后修改时间的函数

`moddatetime()`是一个存储当前时间到`timestamp`字段的触发器。 这对于追踪一个表中指定的行的最后修改时间是有用的。

要使用该函数，创建一个使用这个函数的`BEFORE UPDATE`触发器。 指定一个触发器参数：要修改的字段名。该字段必须是`timestamp` 或`timestamp with time zone` 类型。

在`moddatetime.example`中有一个示例。

F.35. sslinfo

`sslinfo`模块提供关于连接到PostgreSQL 时当前客户端提供的SSL认证的信息。如果当前连接没有使用SSL， 那么该模块是无用的（大部分函数将返回NULL）。

这个扩展不会建立，除非安装时配置带有`--with-openssl`。

F.35.1. 提供的函数

`ssl_is_used() returns boolean`

如果当前到服务器的连接使用了SSL则返回TRUE，否则返回FALSE。

`ssl_version() returns text`

返回SSL连接使用的协议名（如SSLv2，SSLv3，或TLSv1）

`ssl_cipher() returns text`

返回SSL连接使用的密码名称（如DHE-RSA-AES256-SHA）。

`ssl_client_cert_present() returns boolean`

如果当前客户端提供了一个有效的SSL客户端证书到服务器则返回TRUE，否则返回FALSE。（服务器可能或可能没有被配置为需要一个客户端证书。）

`ssl_client_serial() returns numeric`

返回当前客户端证书的序列号。证书序列号和证书发行者的组合保证为唯一的标识一个证书（但不是它的所有者—所有者应该定期的改变它的秘钥，并从发行者处获得新的证书。）

所以，如果你运行自己的CA并且只允许来自这个CA的证书被服务器接受， 那么序列编号是识别一个用户最可靠（虽然不是很帮助记忆）的手段。

`ssl_client_dn() returns text`

返回当前客户端认证的全部科目，转换字符数据为当前数据库编码。 假设如果在证书名里使用非ASCII字符，那么数据库也可以表示这些字符。 如果数据库使用SQL_ASCII编码，那么名字中的非ASCII字符将用UTF-8序列表示。

结果看起来像`/CN=Somebody /C=Some country/O=Some organization`。

`ssl_issuer_dn() returns text`

返回当前客户端认证的全部发行者名字，转换字符数据为当前数据库编码。 编码转换的处理和`ssl_client_dn`相同。

这个函数的返回值的组合和证书序列编号唯一的标识该证书。

只有你有多于一个信任的CA证书在服务器的`root.crt`文件时， 或者如果这个CA已经发布了一些中级证书授权证书，这个函数才真正的有用。

`ssl_client_dn_field(fieldname text) returns text`

这个函数返回证书主题中的指定字段值，或者如果该字段不存在则为NULL。 为字符串常量的字段名转换为使用OpenSSL对象数据库的ASN1对象标识符。 下列的值是可接受的：

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

所有这些字段都是可选的，除了`commonName`。 它完全取决于你的证书的政策将包括或不包括。不过， 这些字段的含义严格的由X.500和X.509标准定义，所以不能任意分配它们的含义。

`ssl_issuer_field(fieldname text) returns text`

和`ssl_client_dn_field`相同，除了是证书发行者而不是证书主题。

F.35.2. 作者

Victor Wagner `<[vitus@cryptocom.ru](mailto:vitus@cryptocom.ru)>`, Cryptocom LTD

Cryptocom OpenSSL开发团队的E-Mail：`<[openssl@cryptocom.ru](mailto:openssl@cryptocom.ru)>`

F.36. tablefunc

`tablefun` 扩展包括了返回表记录(即:多行)的一系列函数．这些函数在数据记录的处理和运用C函数返回多行记录 中是非常有用的

F.36.1. 函数列表

Table F-28 列出了 `tablefunc` 扩展提供的函数

Table F-28. `tablefunc` 函数

函数
`normal_rand(int numvals, float8 mean, float8 stddev)`
`crosstab(text sql)`
`crosstab` *N* `(text sql)`
`crosstab(text source_sql, text category_sql)`
`crosstab(text sql, int N)`
`connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld ], tex`

F.36.1.1. `normal_rand`

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

normal函数返回一系列正态分布的随机值(高斯分布)

numvals是这个函数返回值的数目,mean时正态分布的平均值,stddev是正态分布的方差

例如:这个查询返回1000个均值为5,方差为3的值

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
   -0.369151492880995
    0.283600703686639
        .
        .
        .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.36.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

crosstab函数返回一个二维表,数据在这个表里通过横向而非纵向列出,例如: 我们已有数据如下:

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

而我们想要按如下输出:

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

crosstab函数使用一个SQL查询作为参数,返回一个给定格式的二维表

sql参数用来产生一组数据源,这个语句必须返回1个row_name列,一个category列,一个value列. N是一个废弃的参数,

例如,一个查询需要产生如下形式的数据:

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

crosstab函数返回一系列记录,输出列的实际名字和类型必须定义在FROM字句中,例如:

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

This example produces a set something like:

```
           <== value  columns  ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

FROM字句必须定义一个row_name(和你查询返回第一列具有相同数据类型)列和一个和N个value列 (和查询返回第三列的类型匹配). 由你来定义输出列的名字和数量.

crosstab函数对于具有相同row_name的一组数据返回一行.它使用这些行 从左到右填充输出到二维表.如果这组数据少于输出的列数,则使用null填充, 如果有多余的行数,则额外的输入将被忽略

SQL查询应该一直指定ORDER BY1,2来确保输入的数据时有序的,即, 具有相同row_name的行是连续的并且在一行的输出中是有序的.注意crosstab 不关注第二列查询的结果;它只是用来保证第三列的值在输出中的排序.

这是一个完整的例子:

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

可以通过定义一个crosstab函数去定义任意类型,数量的列,来避免每次在FROM 字句中制定输出列的值和类型.这将在下一节中讲解.另外一种方式是在定义视图的 FROM字句中制定.

F.36.1.3. `crosstab` *N* `(text)`

```
crosstab_N_(text sql)
```

crosstabN函数是crosstab函数封装的一个例子,在使用SELECT查询中,你可以不指定列数 和类型.tablefunc扩展定义了crosstab2,crosstab3,crosstab4函数中列的类型数量.

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

当输入查询返回text类型的row_name并且你有2,3,4列输出值时你可以直接使用 这些函数.其他情况需要像上面的定义crosstab函数一样.

例如,上面的例子也可以像下面一样运行.

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

这些函数大部分只是为了举例说明.你可以按照crosstab函数定义你需要的返回类型的 函数.这里有两种方法来实现:

- 定义一个复合类型的输出列类似于contrib/tablefunc/tablefunc--1.0.sql中的例子. 按照crosstab函数定义的方式,定义一个唯一函数名来接收一个text类型的参数,返回setof your_type_name. 例如,你的数据源返回一个text类型的row_name,values字段为float8类型,并且你需要5列:

```
CREATE TYPE my_crosstab_float8_5_cols AS (
    my_row_name text,
    my_category_1 float8,
    my_category_2 float8,
    my_category_3 float8,
    my_category_4 float8,
    my_category_5 float8
);

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
    RETURNS setof my_crosstab_float8_5_cols
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

- 使用OUT参数定义返回值类型.上面的例子可以像下面定义:

```
CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
    IN text,
    OUT my_row_name text,
    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.36.1.4. `crosstab(text, text)`

```
crosstab(text source_sql, text category_sql)
```

单参数的crosstab函数的主要限制是处理一组记录时插入每一个值到第一个列中. 如果你想插入值到指定的数据属性中,一些数据可能没有该属性的值,这样它将不能很好 的处理.两个参数的crosstab通过提供一个指定的属性列表能较好的处理该类问题.

source_sql语句返回一组数据源.这个语句返回一个row_name列,一个category列 ,和一个value列,它也可能有一个或多个extra列.row_name必须是第一列.category和 value列必须是最后两列.任何在在row_name和category中间的列被认为时extra列. 有相同row_name的extra列被认为是相同的.

例如,source_sql可能返回下面的一系列记录:

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

category_sql返回categories记录.这个语句只能返回一列.它必须最少返回一行,否则 会报错.它也不能返回重复的值,否则会报错.category_sql必须如下面:

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

crosstab函数返回一系列记录,列的名字和类型必须定义在SELECT中的FROM字句中.

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

这将返回如下记录:

```
                  <==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

FROM字句必须定义类型匹配的适当数量的列.如果在source_sql中返回N列,前面的N-2 列必须匹配前N-2列.剩下的列必须和source_sql返回的列类型匹配,并且返回的行数需要和category_sql返回 的相同.

crosstab函数对于具有相同row_name的一组连续记录返回一行输出.复制这组记录 的第一行到row_name和extra列.记录中匹配category的值填充到value列中. 如果一行的category不能匹配任何category_sql返回的结果,它的值将被忽略. 如果输入行中的category不匹配任何category_sql的返回结果,这一个输出列 将被填充为null.

实际上source_sql一直使用ORDER BY 1去确保具有相同row_name的行连续.然而,categories排序不是必须.它实际上时为了保证匹配category_sql的记录有序.

这有两个完整的例子:

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)
```

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate
-------+--------------------------+-------------+-------------+--------------------------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00 2003
(2 rows)
```

你能预定义一个函数来避免在每一个查询中都指定输出列的名字和类型. 查看前面部分的例子. 构成这种形式的crosstab被命名为crosstab_hash.

F.36.1.5. `connectby`

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

connectby函数返回一个表的分层显示.这个表必须有一个键值去唯一的标识一行, 并且有一个父节点键值去关联每一行.connectby能从任意一行列出一个子树

Table F-29 explains the parameters.

Table F-29. `connectby` 参数

参数	描述
`relname`	源表的名字
`keyid_fld`	键值的名字
`parent_keyid_fld`	父节点键值的名字
`orderby_fld`	排序兄弟的字段名字
`start_with`	键值开始值
`max_depth`	最大深度,0表示深度无限
`branch_delim`	每个分支的独立键值

键值和父节点键值可以是相同的任意类型,注意start_with必须是一个字符串,无论key的类型.

connectby函数返回一系列记录,所以输出列的名字和类型必须定义在SELECT的FROM字句中,例如:

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

输出的前两列做为现在行的键值和父节点的键值;他们必须和表的键值类型 匹配.第三列为节点在树中的深度必须为整形.如果给出branch_delim参数,下一列是 一个text类型的分支列,最后如果给出orderby_fld参数最后一列是一个整形的自增值.

branch列给出键值到达当前行的路径.键值被branch_delim字段给出的字符分割开. 如果不想显示分支,在每一个输出列表中忽略branch_delim和branch参数.

如果想要排序同一个父节点的兄弟节点,指定orderby_fld参数来指定需要排序兄弟 节点的字段.如果orderby_fld被指定,输出列一定包括一个整形自增的列.

表和字段值的参数被connectby函数自动的复制到查询语句中.因此如果名字包含特殊 字符或者混合字段请使用"",你可能也需要使用模式来限定表名.

在大的表中,除非在父节点的键值上有一个索引,否则效率将会很差.

在键值中不出现branch_delim参数是很重要的,否则connectby可能错误的报告一个死循环的错误. 如果不使用branch_delim参数,为了递归检查将使用~来作为默认值.

下面是一个例子:

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+---------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- without branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.36.2. Author

Joe Conway

F.37. tcn

The `tcn` module provides a trigger function that notifies listeners of changes to any table on which it is attached. It must be used as an `AFTER` trigger `FOR EACH ROW`.

Only one parameter may be supplied to the function in a `CREATE TRIGGER` statement, and that is optional. If supplied it will be used for the channel name for the notifications. If omitted `tcn` will be used for the channel name.

The payload of the notifications consists of the table name, a letter to indicate which type of operation was performed, and column name/value pairs for primary key columns. Each part is separated from the next by a comma. For ease of parsing using regular expressions, table and column names are always wrapped in double quotes, and data values are always wrapped in single quotes. Embedded quotes are doubled.

A brief example of using the extension follows.

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute procedure triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'" recei
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'" recei
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'" recei
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'" recei
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'" recei
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'" recei
```

F.38. test_parser

`test_parser` 是一个自定义的全文搜索解析器的例子。它不做特别有用的工作，但是可以作为开发一个自己的解析器的起点。

`test_parser` 识别以空格分隔的单词，并且返回两种令牌类型：

```
mydb=# SELECT * FROM ts_token_type('testparser');
 tokid | alias |  description
-------+-------+---------------
     3 | word  | Word
    12 | blank | Space symbols
(2 rows)
```

这些令牌数字被选择来兼容默认的解析器编号。这允许我们可以使用 `headline()` 函数，从而保持例子的简单。

F.38.1. 用法

安装 `test_parser` 扩展，创建一个文本搜索的解析器testparser。它没有用户可配置参数。

你可以像下面的例子这样测试该解析器，

```
mydb=# SELECT * FROM ts_parse('testparser', 'That''s my first own parser');
 tokid | token
-------+--------
     3 | That's
    12 |
     3 | my
    12 |
     3 | first
    12 |
     3 | own
    12 |
     3 | parser
```

实际使用需要配置一个文本搜索配置项来使用这个解析器。例如，

```
mydb=# CREATE TEXT SEARCH CONFIGURATION testcfg ( PARSER = testparser );
CREATE TEXT SEARCH CONFIGURATION

mydb=# ALTER TEXT SEARCH CONFIGURATION testcfg
mydb-#   ADD MAPPING FOR word WITH english_stem;
ALTER TEXT SEARCH CONFIGURATION

mydb=#  SELECT to_tsvector('testcfg', 'That''s my first own parser');
          to_tsvector
-------------------------------
 'that':1 'first':3 'parser':5
(1 row)

mydb=# SELECT ts_headline('testcfg', 'Supernovae stars are the brightest phenomena in gal
mydb(#                    to_tsquery('testcfg', 'star'));
                           ts_headline
-----------------------------------------------------------------
 Supernovae <b>stars</b> are the brightest phenomena in galaxies
(1 row)
```

F.39. tsearch2

The tsearch2 module provides backwards-compatible text search functionality for applications that used tsearch2 before text searching was integrated into core PostgreSQL in release 8.3.

F.39.1. Portability Issues

Although the built-in text search features were based on tsearch2 and are largely similar to it, there are numerous small differences that will create portability issues for existing applications:

- Some functions' names were changed, for example `rank` to `ts_rank`. The replacement `tsearch2` module provides aliases having the old names.
- The built-in text search data types and functions all exist within the system schema `pg_catalog`. In an installation using tsearch2, these objects would usually have been in the `public` schema, though some users chose to place them in a separate schema of their own. Explicitly schema-qualified references to the objects will therefore fail in either case. The replacement `tsearch2` module provides alias objects that are stored in `public` (or another schema if necessary) so that such references will still work.
- There is no concept of a "current parser" or "current dictionary" in the built-in text search features, only of a current search configuration (set by the `default_text_search_config` parameter). While the current parser and current dictionary were used only by functions intended for debugging, this might still pose a porting obstacle in some cases. The replacement `tsearch2` module emulates these additional state variables and provides backwards-compatible functions for setting and retrieving them.

There are some issues that are not addressed by the replacement `tsearch2` module, and will therefore require application code changes in any case:

- The old `tsearch2` trigger function allowed items in its argument list to be names of functions to be invoked on the text data before it was converted to `tsvector` format. This was removed as being a security hole, since it was not possible to guarantee that the function invoked was the one intended. The recommended approach if the data must be massaged before being indexed is to write a custom trigger that does the work for itself.

- Text search configuration information has been moved into core system catalogs that are noticeably different from the tables used by tsearch2. Any applications that examined or modified those tables will need adjustment.
- If an application used any custom text search configurations, those will need to be set up in the core catalogs using the new text search configuration SQL commands. The replacement `tsearch2` module offers a little bit of support for this by making it possible to load an old set of tsearch2 configuration tables into PostgreSQL 8.3. (Without the module, it is not possible to load the configuration data because values in the `regprocedure` columns cannot be resolved to functions.) While those configuration tables won't actually *do* anything, at least their contents will be available to be consulted while setting up an equivalent custom configuration in 8.3.
- The old `reset_tsearch()` and `get_covers()` functions are not supported.
- The replacement `tsearch2` module does not define any alias operators, relying entirely on the built-in ones. This would only pose an issue if an application used explicitly schema-qualified operator names, which is very uncommon.

F.39.2. Converting a pre-8.3 Installation

The recommended way to update a pre-8.3 installation that uses tsearch2 is:

1. Make a dump from the old installation in the usual way, but be sure not to use `-c` (`--clean`) option of pg_dump or pg_dumpall.
2. In the new installation, create empty database(s) and install the replacement `tsearch2` module into each database that will use text search. This must be done *before* loading the dump data! If your old installation had the tsearch2 objects in a schema other than `public` , be sure to adjust the `CREATE EXTENSION` command so that the replacement objects are created in that same schema.
3. Load the dump data. There will be quite a few errors reported due to failure to recreate the original tsearch2 objects. These errors can be ignored, but this means you cannot restore the dump in a single transaction (eg, you cannot use pg_restore's `-1` switch).
4. Examine the contents of the restored tsearch2 configuration tables (`pg_ts_cfg` and so on), and create equivalent built-in text search configurations as needed. You may drop the old configuration tables once you've extracted all the useful information from them.
5. Test your application.

At a later time you may wish to rename application references to the alias text search objects, so that you can eventually uninstall the replacement `tsearch2` module.

F.39.3. References

Tsearch2 Development Site http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.40. unaccent

`unaccent` 是一个文本搜索字典，它从词汇中去掉重音符号（变音标志符号）。 这是一个过滤词典，这意味着它的输出总是传递给下一个字典（如果存在的话），而不像常规行为的字典。 这允许对全文搜索进行重音不敏感的处理。

`unaccent` 的当前实现不能用作一个 `thesaurus` 字典的规范字典。

F.40.1. 配置

一个 `unaccent` 字典接受下面的操作：

- `RULES` 是包含翻译规则列表的文件的基本名称。 这个文件必须存储在 `$SHAREDIR/tsearch_data/` 文件夹下 (`$SHAREDIR` 是指 PostgreSQL安装时的数据共享文件)。 它的名字必须以 `.rules` 为后缀 (这个后缀无须出现在 `RULES` 参数中)。

规则文件的格式如下：

- 每一个行代表一个字符对，由带重音符号的字符跟着一个不带重音符号的字符组成。第一个字符将被翻译为第二个字符。例如：

```
À        A
Á        A
Â        A
Ã        A
Ä        A
Å        A
Æ        A
```

一个对大多数欧洲语言都直接有用的更完整的例子，可以在 `unaccent.rules` 中找到，这个文件在 `unaccent` 模块被安装时就被置于 `$SHAREDIR/tsearch_data/` 文件夹下。

F.40.2. 用法

在安装 `unaccent` 扩展的时候，会创建一个 `unaccent` 文本搜索模版和一个基于这个模版的字典。 这个 `unaccent` 字典有一个默认的参数设置 `RULES='unaccent'` ，这使得标准的 `unaccent.rules` 配置可以立即生效。 如果有需要，你也可以改变这个参数，例如，

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

或者创建基于 `unaccent` 模版的新字典。

你可以试试下面的sql，来测试这个字典，

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

这里有一个例子，演示如何将 `unaccent` 字典增加到文本搜索配置中：

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
(1 row)
```

F.40.3. 函数

`unaccent()` 函数从一个给定的字符串中去掉重音符号（变音标志符号）。基本上，它是一个 `unaccent` 字典的包装，但它可以超出正常文本搜索的上下文使用。

```
unaccent([`_dictionary_`, ] _string_) returns text
```

例如，

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.41. uuid-ossp

`uuid-ossp` 模块提供了一些函数用来生成通用唯一识别码(UUID)， 它支持几种 UUID 产生的标准算法。同时它还提供了一些函数用来产生某些特定 的UUID常量。

这个模块依赖于 OSSP UUID 库， http://www.ossp.org/pkg/lib/uuid/.

F.41.1. `uuid-ossp` 函数

Table F-30 中的函数用来产生 UUID。 相关标准 ITU-T Rec. X.667, ISO/IEC 9834-8:2005 和 RFC 4122 定义了四种生成 UUID 的算法, 分别在版本 1, 3, 4 和5中定义(没用版本2算法)每个算法适合于不同种类的应用使用。

Table F-30. UUID 生成函数

函数	描述
`uuid_generate_v1()`	这个函数生成版本1的 UUID。它的算法使戳。注意这种 UUID 泄露了生成它的计算它可能不太适合对安全性要求较高的应用
`uuid_generate_v1mc()`	这个函数生成一个版本1的 UUID，但是使不是计算机的真实的 MAC 地址。
`uuid_generate_v3(namespace uuid, name text)`	这个函数使用给定的输入名字(name)在给生成一个版本3的 UUID。给定的命名空间的函数 `uuid_ns_*()` 返回的常量。(理论上 name 是一个选定命名空间(namespace)中 `SELECT uuid_generate_v3(uuid_ns_url(),` 参数 name会被使用 MD5 算法做哈希，所向获得明文。 利用这个方法生成的 UUID 行相关的环境因素，因此生成过程是可重
`uuid_generate_v4()`	这个函数生成一个版本4的 UUID，它完全
`uuid_generate_v5(namespace uuid, name text)`	这个函数生成一个版本5的 UUID，它个工 UUID，但是它使用的 SHA-1 的哈希算法 MD5 算法更安全，所有应该尽量使用版本

Table F-31. 返回 UUID 常量的函数

`uuid_nil()`	**一个 "nil" UUID 常量，它不应该看作一个真正的 UUID。**
`uuid_ns_dns()`	代表 DNS 命令空间的 UUID 常量。
`uuid_ns_url()`	代表 URL 命名空间的 UUID 常量。
`uuid_ns_oid()`	代表 ISO 对象标识符(OID)命名空间的 UUID 常量。(它是 ASN.1 的 OID，和 PostgreSQL 用的 OID 没有关系）。
`uuid_ns_x500()`	代表 X.500 识别名字(DN)命名空间的 UUID 常量

F.41.2. 作者

Peter Eisentraut `<[peter_e@gmx.net](mailto:peter_e@gmx.net)>`

F.42. xml2

`xml2` 模块提供XPath查询和XSLT功能。

F.42.1. 弃用通知

从PostgreSQL 8.3开始，在内核服务器中有一个基于SQL/XML标准的XML相关的功能。 这个功能包含了XML语法检查和XPath查询，这也是这个模块的包含的，甚至更多，不过API一点也不兼容。 计划在未来的PostgreSQL版本中删除这个模块，支持新的标准的API，所以鼓励你尝试转换你的应用。 如果你发现这个模块的某些功能在一个适当形式的更新的API中不可用， 请解释你的问题到 `<[pgsql-hackers@postgresql.org](mailto:pgsql-hackers@postgresql.org)>` ，以便解决这个缺陷。

F.42.2. 功能说明

Table F-32显示了这个模块提供的函数。 这些函数提供了简单的XMP解析和XPath查询。所有参数都是 `text` 类型的， 所以为了简化没有显示。

Table F-32. 函数

函数	返回	
`xml_is_well_formed(document)`	`bool`	解析它的参数中的文件正 真。 （注意：在PostgreS 是错误的名字，因为在XM 仍然可用，但是已经废弃
`xpath_string(document, query)`	`text`	这个函数在提供的文件上
`xpath_number(document, query)`	`float4`	
`xpath_bool(document, query)`	`bool`	

`xpath_nodeset(document, query, toptag, itemtag)`	`text`	在文件上评估查询并将结
		么输出将看起来像： `<`
		`<itemtag>Value 1 v`
		`<itemtag>Value 2..`
		果 `toptag` 或 `itemtag` 中的

`xpath_nodeset(document, query)`	`text`	类似于 `xpath_nodeset(doc` 略两个标签。
`xpath_nodeset(document, query, itemtag)`	`text`	类似于 `xpath_nodeset(doc` 略toptag。
`xpath_list(document, query, separator)`	`text`	这个函数返回由指定的分 回值是 `Value 1,Value 2,V`
`xpath_list(document, query)`	`text`	这是一个对于使用 `,` 作为

F.42.3. `xpath_table`

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns s
```

`xpath_table` 是一个表函数，在每一组文档上计算一组XPath查询， 并作为一个表返回结果。 原始文档表的主键字段作为结果的第一个字段返回， 因此结果集可以容易的用于连接。参数在Table F-33中描述。

Table F-33. `xpath_table` 参数

参数	描述
`key`	"key"字段的名字—这只是用作输出表的第一个字段，也就是， 它识别每个传来的输出行的记录（请参见下面关于多行值的注释）
`document`	包含XML文档的字段名
`relation`	包含文档的表或视图的名字
`xpaths`	一个或多个XPath表达式，由 `|` 分隔
`criteria`	WHERE子句的内容。这个不能省略，所以如果你想要在关系中处理所有的行， 那么使用 `true` 或 `1=1`

这些参数（除了XPath字符串）只是代替一个纯SQL SELECT语句， 所以你有一些灵活性—该语句是

```
SELECT &lt;key&gt;, &lt;document&gt; FROM &lt;relation&gt; WHERE &lt;criteria&gt;
```

所以这些参数可以是任何在这些特定位置的有效值。 这个SELECT的结果需要返回正好两个字段（除非你尝试为key或document列出多个字段）。 注意这个简单的方法需要验证任何用户提供的值，以避免SQL注入攻击。

该函数必须用于 `FROM` 表达式中，用一个 `AS` 子句指定输出字段；例如

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

`AS` 子句定义输出表中字段的名字和类型。第一个是"key"字段， 剩余的对应于XPath查询。如果XPath查询多于结果字段，那么额外的查询将被忽略。 如果结果字段多于XPath查询，那么额外的字段将为空。

请注意，这个示例定义 `page_count` 的结果字段为一个整数。 该函数内部处理字符串的表示，所以当你说你在输出中想要一个整数时， 它将接收XPath结果的字符串表示，并使用PostgreSQL输入函数将它转换为一个整数 （或者任何 `AS` 子句要求的类型）。如果它不能这么做则导致一个错误— 例如，如果结果为空—所以如果你认为你的数据有任何问题， 你可能想要坚持 `text` 作为该字段的类型。

调用的 `SELECT` 语句不需要只是 `SELECT *` — 它可以通过名字或链接输出字段到其他的表引用输出字段。 该函数产生一个虚拟表，用该虚拟表你可以执行任何你想要的操作 （例如，聚集、链接、排序等等）。所以我们也可以将：

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

作为一个更复杂的示例。当然，为了方便，你可以将所有这些包装到一个视图中。

F.42.3.1. 多值的结果

`xpath_table` 函数假设每个XPath查询的结果可能是多值的， 所以该函数返回的行数可能和输入文档的行数不同。 返回的第一行包含每个查询的第一个结果，第二行包含每个查询的第二个结果。 如果其中的一个查询比其他查询的值少，那么将返回空值。

在某些情况下，用户知道一个给定的XPath查询将只返回一个结果（可能是一个唯一文档标识符）— 如果和返回多个结果的XPath查询一起使用，那么唯一值结果将只在结果的第一行出现。 这个问题的解决方法是使用该关键字段作为一个更简单的XPath查询连接的一部分。 示例：

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

要在每个行上获得 `doc_num`，解决方法是使用两个 `xpath_table` 的调用并连接结果：

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.42.4. XSLT 函数

如果安装了libxslt，那么下列的函数是可用的：

F.42.4.1. `xslt_process`

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

这个函数应用XSL样式表到文档并返回转换了的结果。 `paramlist` 是一个在转换中使用的参数分配列表，以格式`a=1,b=2`指定。 请注意，参数分析是非常简单的：参数值不能包含逗号！

也有一个两参数的`xslt_process`版本，不传递任何参数到转换。

F.42.5. 作者

John Gray `<[jgray@azuli.co.uk](mailto:jgray@azuli.co.uk)>`

这个模块的发展是由Torchbox Ltd. (www.torchbox.com)赞助的。 它和PostgreSQL有一样的BSD许可证。

Appendix G. 额外提供的程序

Table of Contents

本附录和前一个包含 可以在PostgreSQL发布的`contrib`目录中发现 的有关模块的信息。 参阅Appendix F获取更多关于`contrib`部分和 服务器扩展以及特别在`contrib`中发现的插件的更多详细信息。

本附录包括在`contrib`中发现的实用程序。 一旦安装，无论来自源或包装系统， 它们在PostgreSQL安装的`bin`目录中被发现 可用于类似的任何其他程序。

G.1. 客户端应用程序

Table of Contents

这部分在`contrib`中 包含PostgreSQL客户端 应用程序。它们可以从任何地方允许， 数据库服务器独立的地方。参阅Reference II, *PostgreSQL 客户端应用程序*获取 关于核心PostgreSQL发布 部分客户端应用程序的信息。

oid2name

Name

oid2name -- 解析PostgreSQL 数据目录里的 OID和文件节点

Synopsis

`oid2name` [`_选项_` ...]

描述

oid2nameoid2name 是一个帮助管理员检查PostgreSQL使用的文件结构的工具程序。要使用这个工具，你必须熟悉数据库文件结构，在第58章描述 Chapter 58.

> **Note:** "oid2name" 很有历史了,但是实际上有相当的误导，,因为大多数你使用它的时候,你将真正和的‘filenode numbers’(他们的文件名在数据库目录中是可见的)连接。 请确保你明白表的OID和表filenodes之间的区别

oid2name 连接到一个目标数据库并且提取OID，文件节点，和/或表名信息 你也可以展示数据库的OID和表空间的OID

选项

oid2name接收下面的命令行参数:

`-f` `_filenode_`

展示文件节点是 `_filenode_` 的表的信息

`-i`

在这个列表中包含索引和序列

`-o` `_oid_`

展示OID是 `_oid_` 的表的信息

`-q`

省略表头(对脚本很有用)

`-s`

展示表空间的OID

`-S`

包含在(`information_schema` , `pg_toast` and `pg_catalog` schemas)里的系统对象

`-t` `_tablename_pattern_`

展示匹配 `_tablename_pattern_` 的表信息

`-V``--version`

打印 oid2name 版本并退出.

`-x`

列出每一个被展示的对象的更多信息: tablespace name, schema name, and OID

`-?` `--help`

显示关于 oid2name命令行的帮助信息和参数，并退出

oid2name在连接的时候也接受下面的命令行参数:

`-d` `_database_`

连接哪台数据库

`-H` `_host_`

数据库服务器的地址

`-p` `_port_`

数据库服务器的端口

`-U` `_username_`

连接数据库的用户

`-P` `_password_`

密码（但是不建议这么做，因为把密码放在命令行存在安全隐患）

要展示特定的表, 选择要展示的数据库使用 `-o` , `-f` 和/或 `-t` . `-o` 需要一个 OID, `-f` 需要一个 filenode, `-t` 需要一个表名 (事实上，它是一个 `LIKE` 模式的, 所以你可以使用类似 `foo%` 的参数). 只要你喜欢，你可以使用更多的选项，列表将包含所有的匹配这些选项的对象. 但是请注意这些选项只能展示 `-d` 指派的数据库里的对象.

如果你没有提供 `-o` , `-f` or `-t` 中的任何参数, 但是给了 `-d` 参数, 它将列出以 `-d` 指派的数据库的所有的表. 在这种模式下， `-S` 和 `-i` 选项控制着列表的显示内容.

如果你也没有指定 `-d` 它将展示数据库列表的 OID. 或者你可以通过 `-s` 参数去得到表空间的列表

注意

oid2name需要一个正在运行的且系统的catalogs没有被破坏的数据库服务器，因此，从遭受灾难性破坏的数据库中恢复数据，将会很有限。

例子

```
$ # 数据库服务器上都有哪些数据库?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # 好，让我们进入alvherre数据库的目录
$ cd $PGDATA/base/17228

$ # 获取缺省表空间的前十个数据库对象，并且以size排序
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ #我想知道155173这个文件是什么
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # 你也可以请求更多的对象
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
    155173       accounts
   1155291  accounts_pkey

$ #你可以混合这些选项，通过-x获取更多的详细信息
$ oid2name -d alvherre -t accounts -f 1155291 -x
```

```
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
------------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # 展示每个数据库对象所占的磁盘空间
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # 和上面一样，但是按照大小排序
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561            1155291  accounts_pkey
1177              16717  pg_proc_proname_args_nsp_index
...

$ # 如果你想看表空间里有什么，使用 pg_tblspc 目录
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ #数据库在 "fastdisk"表空间都有什么？ ?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # 这个数据库 17228 又是什么?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # 让我们看看这个数据库在表空间里都有什么
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # 这个是一个很不错的小表，但是它是什么表呢?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

作者

B. Palmer `<[bpalmer@crimelabs.net](mailto:bpalmer@crimelabs.net)>`

pgbench

Name

pgbench -- run a benchmark test on PostgreSQL

Synopsis

`pgbench` `-i` [`_option_` ...] [`_dbname_`]

`pgbench` [`_option_` ...] [`_dbname_`]

Description

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five `SELECT`, `UPDATE`, and `INSERT` commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

Typical output from pgbench looks like:

```
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

The first six lines report some of the most important parameter settings. The next line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions per client); these will be equal unless the run failed before completion. (In `-T` mode, only the actual number of transactions is printed.) The last two lines report the number of transactions per second, figured with and without counting the time to start database sessions.

The default TPC-B-like transaction test requires specific tables to be set up beforehand. pgbench should be invoked with the `-i` (initialize) option to create and populate these tables. (When you are testing a custom script, you don't need this step, but will instead need to do whatever setup your test needs.) Initialization looks like:

```
pgbench -i [ `_other-options_` ] _dbname_
```

where `_dbname_` is the name of the already-created database to test in. (You may also need `-h`, `-p`, and/or `-U` options to specify how to connect to the database server.)

Caution
`pgbench -i` creates four tables `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`, destroying any existing tables of these names. Be very careful to use another database if you have tables having these names!

At the default "scale factor" of 1, the tables initially contain this many rows:

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

You can (and, for most purposes, probably should) increase the number of rows by using the `-s` (scale factor) option. The `-F` (fillfactor) option might also be used at this point.

Once you have done the necessary setup, you can run your benchmark with a command that doesn't include `-i`, that is

```
pgbench [ `_options_` ] _dbname_
```

In nearly all cases, you'll need some options to make a useful test. The most important options are `-c` (number of clients), `-t` (number of transactions), `-T` (time limit), and `-f` (specify a custom script file). See below for a full list.

Options

The following is divided into three subsections: Different options are used during database initialization and while running benchmarks, some options are useful in both cases.

Initialization Options

pgbench accepts the following command-line initialization arguments:

`-i`

Required to invoke initialization mode.

`-F` `_fillfactor_`

Create the `pgbench_accounts` , `pgbench_tellers` and `pgbench_branches` tables with the given fillfactor. Default is 100.

`-n`

Perform no vacuuming after initialization.

`-q`

Switch logging to quiet mode, producing only one progress message per 5 seconds. The default logging prints one message each 100000 rows, which often outputs many lines per second (especially on good hardware).

`-s` `_scale_factor_`

Multiply the number of rows generated by the scale factor. For example, `-s 100` will create 10,000,000 rows in the `pgbench_accounts` table. Default is 1. When the scale is 20,000 or larger, the columns used to hold account identifiers (`aid` columns) will switch to using larger integers (`bigint`), in order to be big enough to hold the range of account identifiers.

`--foreign-keys`

Create foreign key constraints between the standard tables.

`--index-tablespace=``_index_tablespace_`

Create indexes in the specified tablespace, rather than the default tablespace.

`--tablespace=``_tablespace_`

Create tables in the specified tablespace, rather than the default tablespace.

`--unlogged-tables`

Create all tables as unlogged tables, rather than permanent tables.

Benchmarking Options

pgbench accepts the following command-line benchmarking arguments:

`-c` `_clients_`

Number of clients simulated, that is, number of concurrent database sessions. Default is 1.

`-C`

Establish a new connection for each transaction, rather than doing it just once per client session. This is useful to measure the connection overhead.

`-d`

Print debugging output.

`-D` `_varname_``=``_value_`

Define a variable for use by a custom script (see below). Multiple `-D` options are allowed.

`-f` `_filename_`

Read transaction script from `_filename_`. See below for details. `-N`, `-S`, and `-f` are mutually exclusive.

`-j` `_threads_`

Number of worker threads within pgbench. Using more than one thread can be helpful on multi-CPU machines. The number of clients must be a multiple of the number of threads, since each thread is given the same number of client sessions to manage. Default is 1.

`-l`

Write the time taken by each transaction to a log file. See below for details.

`-M` `_querymode_`

Protocol to use for submitting queries to the server:

- `simple`: use simple query protocol.
- `extended`: use extended query protocol.
- `prepared`: use extended query protocol with prepared statements.

The default is simple query protocol. (See Chapter 48 for more information.)

`-n`

Perform no vacuuming before running the test. This option is *necessary* if you are running a custom test scenario that does not include the standard tables `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`.

`-N`

Do not update `pgbench_tellers` and `pgbench_branches`. This will avoid update contention on these tables, but it makes the test case even less like TPC-B.

`-r`

Report the average per-statement latency (execution time from the perspective of the client) of each command after the benchmark finishes. See below for details.

`-s` `_scale_factor_`

Report the specified scale factor in pgbench's output. With the built-in tests, this is not necessary; the correct scale factor will be detected by counting the number of rows in the `pgbench_branches` table. However, when testing custom benchmarks (`-f` option), the scale factor will be reported as 1 unless this option is used.

`-S`

Perform select-only transactions instead of TPC-B-like test.

`-t` `_transactions_`

Number of transactions each client runs. Default is 10.

`-T` `_seconds_`

Run the test for this many seconds, rather than a fixed number of transactions per client. `-t` and `-T` are mutually exclusive.

`-v`

Vacuum all four standard tables before running the test. With neither `-n` nor `-v` , pgbench will vacuum the `pgbench_tellers` and `pgbench_branches` tables, and will truncate `pgbench_history` .

`--aggregate-interval=``_seconds_`

Length of aggregation interval (in seconds). May be used only together with -l - with this option, the log contains per-interval summary (number of transactions, min/max latency and two additional fields useful for variance estimation).

This option is not currently supported on Windows.

`--sampling-rate=``_rate_`

Sampling rate, used when writing data into the log, to reduce the amount of log generated. If this option is given, only the specified fraction of transactions are logged. 1.0 means all transactions will be logged, 0.05 means only 5% of the transactions will be logged.

Remember to take the sampling rate into account when processing the log file. For example, when computing tps values, you need to multiply the numbers accordingly (e.g. with 0.01 sample rate, you'll only get 1/100 of the actual tps).

Common Options

pgbench accepts the following command-line common arguments:

`-h` `_hostname_`

The database server's host name

`-p` `_port_`

The database server's port number

`-U` `_login_`

The user name to connect as

`` -V``--version ``

Print the pgbench version and exit.

`-?` `--help`

Show help about pgbench command line arguments, and exit.

Notes

What is the "Transaction" Actually Performed in pgbench?

The default transaction script issues seven commands per transaction:

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`
3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`
6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :d`
7. `END;`

If you specify `-N`, steps 4 and 5 aren't included in the transaction. If you specify `-S`, only the `SELECT` is issued.

Custom Scripts

pgbench has support for running custom benchmark scenarios by replacing the default transaction script (described above) with a transaction script read from a file (`-f` option). In this case a "transaction" counts as one execution of a script file. You can even specify multiple scripts (multiple `-f` options), in which case a random one of the scripts is chosen each time a client session starts a new transaction.

The format of a script file is one SQL command per line; multiline SQL commands are not supported. Empty lines and lines beginning with `--` are ignored. Script file lines can also be "meta commands", which are interpreted by pgbench itself, as described below.

There is a simple variable-substitution facility for script files. Variables can be set by the command-line `-D` option, explained above, or by the meta commands explained below. In addition to any variables preset by `-D` command-line options, the variable `scale` is preset to the current scale factor. Once set, a variable's value can be inserted into a SQL command by writing ``:``_variablename_``. When running more than one client session, each session has its own set of variables.

Script file meta commands begin with a backslash (`\`). Arguments to a meta command are separated by white space. These meta commands are supported:

`\set` `_varname_` `_operand1_` [`_operator_` `_operand2_`]

Sets variable `_varname_` to a calculated integer value. Each `_operand_` is either an integer constant or a ``:``_variablename_`` reference to a variable having an integer value. The `_operator_` can be `+` , `-` , `*` , or `/` .

Example:

```
\set ntellers 10 * :scale
```

`\setrandom` `_varname_` `_min_` `_max_`

Sets variable `_varname_` to a random integer value between the limits `_min_` and `_max_` inclusive. Each limit can be either an integer constant or a ``:``_variablename_`` reference to a variable having an integer value.

Example:

```
\setrandom aid 1 :naccounts
```

`\sleep` `_number_` [us | ms | s]

Causes script execution to sleep for the specified duration in microseconds (`us`), milliseconds (`ms`) or seconds (`s`). If the unit is omitted then seconds are the default. `_number_` can be either an integer constant or a ``:``_variablename_`` reference to a variable

having an integer value.

Example:

```
\sleep 10 ms
```

`\setshell` `_varname_` `_command_` [`_argument_` ...]

Sets variable `_varname_` to the result of the shell command `_command_`. The command must return an integer value through its standard output.

`_argument_` can be either a text constant or a ``:``_variablename_`` reference to a variable of any types. If you want to use `_argument_` starting with colons, you need to add an additional colon at the beginning of `_argument_`.

Example:

```
\setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_w
```

`\shell` `_command_` [`_argument_` ...]

Same as `\setshell`, but the result is ignored.

Example:

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

As an example, the full definition of the built-in TPC-B-like transaction is:

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delt
END;
```

This script allows each iteration of the transaction to reference different, randomly-chosen rows. (This example also shows why it's important for each client session to have its own variables — otherwise they'd not be independently touching different rows.)

Per-Transaction Logging

With the `-l` option but without the `--aggregate-interval`, pgbench writes the time taken by each transaction to a log file. The log file will be named `pgbench_log.``_nnn_`, where `_nnn_` is the PID of the pgbench process. If the `-j` option is 2 or higher, creating multiple worker threads, each will have its own log file. The first worker will use the same name for its log file as in the standard single worker case. The additional log files for the other workers will be named `pgbench_log.``_nnn_`. `_mmm_`, where `_mmm_` is a sequential number for each worker starting with 1.

The format of the log is:

```
_client_id_ _transaction_no_ _time_ _file_no_ _time_epoch_ _time_us_
```

where `_time_` is the total elapsed transaction time in microseconds, `_file_no_` identifies which script file was used (useful when multiple scripts were specified with `-f`), and `_time_epoch_` / `_time_us_` are a UNIX epoch format timestamp and an offset in microseconds (suitable for creating a ISO 8601 timestamp with fractional seconds) showing when the transaction completed.

Here are example outputs:

```
 0 199 2241 0 1175850568 995598
 0 200 2465 0 1175850568 998079
 0 201 2513 0 1175850569 608
 0 202 2038 0 1175850569 2663
```

When running a long test on hardware that can handle a lot of transactions, the log files can become very large. The `--sampling-rate` option can be used to log only a random sample of transactions.

Aggregated Logging

With the `--aggregate-interval` option, the logs use a bit different format:

```
_interval_start_ _num_of_transactions_ _latency_sum_ _latency_2_sum_ _min_latency_ _max_l
```

where `_interval_start_` is the start of the interval (UNIX epoch format timestamp), `_num_of_transactions_` is the number of transactions within the interval, `_latency_sum_` is a sum of latencies (so you can compute average latency easily). The following two fields are useful for variance estimation - `_latency_sum_` is a sum of latencies and `_latency_2_sum_` is

a sum of 2nd powers of latencies. The last two fields are `_min_latency_` - a minimum latency within the interval, and `_max_latency_` - maximum latency within the interval. A transaction is counted into the interval when it was committed.

Here is example outputs:

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

Notice that while the plain (unaggregated) log file contains index of the custom script files, the aggregated log does not. Therefore if you need per script data, you need to aggregate the data on your own.

Per-Statement Latencies

With the `-r` option, pgbench collects the elapsed transaction time of each statement executed by every client. It then reports an average of those values, referred to as the latency for each statement, after the benchmark has finished.

For the default script, the output will look similar to this:

```
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
        0.004386        \set nbranches 1 * :scale
        0.001343        \set ntellers 10 * :scale
        0.001212        \set naccounts 100000 * :scale
        0.001310        \setrandom aid 1 :naccounts
        0.001073        \setrandom bid 1 :nbranches
        0.001005        \setrandom tid 1 :ntellers
        0.001078        \setrandom delta -5000 5000
        0.326152        BEGIN;
        0.603376        UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE ai
        0.454643        SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        5.528491        UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid
        7.335435        UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bi
        0.371851        INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
        1.212976        END;
```

If multiple script files are specified, the averages are reported separately for each script file.

Note that collecting the additional timing information needed for per-statement latency computation adds some overhead. This will slow average execution speed and lower the computed TPS. The amount of slowdown varies significantly depending on platform and hardware. Comparing average TPS values with and without latency reporting enabled is a good way to measure if the timing overhead is significant.

Good Practices

It is very easy to use pgbench to produce completely meaningless numbers. Here are some guidelines to help you get useful results.

In the first place, *never* believe any test that runs for only a few seconds. Use the `-t` or `-T` option to make the run last at least a few minutes, so as to average out noise. In some cases you could need hours to get numbers that are reproducible. It's a good idea to try the test run a few times, to find out if your numbers are reproducible or not.

For the default TPC-B-like test scenario, the initialization scale factor (`-s`) should be at least as large as the largest number of clients you intend to test (`-c`); else you'll mostly be measuring update contention. There are only `-s` rows in the `pgbench_branches` table, and every transaction wants to update one of them, so `-c` values in excess of `-s` will undoubtedly result in lots of transactions blocked waiting for other transactions.

The default test scenario is also quite sensitive to how long it's been since the tables were initialized: accumulation of dead rows and dead space in the tables changes the results. To understand the results you must keep track of the total number of updates and when vacuuming happens. If autovacuum is enabled it can result in unpredictable changes in measured performance.

A limitation of pgbench is that it can itself become the bottleneck when trying to test a large number of client sessions. This can be alleviated by running pgbench on a different machine from the database server, although low network latency will be essential. It might even be useful to run several pgbench instances concurrently, on several client machines, against the same database server.

vacuumlo

Name

vacuumlo -- 从 PostgreSQL 数据库移除孤立的大对象

Synopsis

`vacuumlo` [`_option_` ...] `_dbname_` ...

描述

vacuumlo 是一个 "会" 从 PostgreSQL 数据库移除所有孤立的大对象的简单程序. 一个大对象(LO)被认为是OID不在任何数据库的数据列的 `oid` 或者or `lo` 中的LO.

当摸使用它, 你可能会对 lo 模块中的 `lo_manage` 触发器感兴趣. `lo_manage` 在一开始尝试避免创造孤立的LOs是非常有用的.

命令行中出现的数据库都是处理过的.

选项

vacuumlo 接受下面的命令行参数:

`-l` `_限制_`

每个事务移除不超过 `_限制_` 数量的大对象 (默认 1000). 如果服务每移除一个LO就获得一个锁, 在一个事务中移除太多的LOs是非常危险的 max_locks_per_transaction. 如果你想要所有移除在一个单独的事务设置限制为0.

`-n`

不移除任何东西,只是显示什么将要被执行.

`-v`

输出一系列的进度信息.

`-V``--version`

打印 vacuumlo 版本并且退出.

`-?` `--help`

显示 vacuumlo 的命令行参数, 并且退出.

vacuumlo 也接受下面的命令行参数作为连接参:

`-h` `_主机名_`

数据库服务器的主机.

`-p` `_端口_`

数据库服务器的端口.

`-U` `_用户名_`

作为连接服务器的用户名.

`-w` `--no-password`

从来不发出密码提示. 如果服务器要求密码认证并且通过其他方法比如 `.pgpass` 文件中不能获取, 连接尝试将会失败. 在批处理任务或脚本中没有用户输入密码的时候这个选项将会变得很有用.

`-W`

在连接数据库之前强制 vacuumlo 提示密码输入.

这个选项不是必要的, 因为 vacuumlo 将会自动的提示密码输入如果服务器需要密码认证的话. 然而, vacuumlo 将会浪费一个连接尝试去找出服务器是否需要密码. 在某些情况下使用 `-W` 选项来避免额外的连接尝试是值得的.

说明

vacuumlo 以下面的方法工作: 首先, vacuumlo 创建一个包含所有选择数据库的大对象OIDs的临时表. 然后扫描数据库中是 `oid` 或者 `lo` 类型的所有列, 然后从临时表中移除匹配的记录. (注意: 只有这些名字的类型才被考虑; 特别是, 超过它们范围的不被考虑.) 临时表中剩下的记录被认证维LOs对象.它们将会被移除.

作者

Peter Mount `<[peter@retep.org.uk](mailto:peter@retep.org.uk)>`

G.2. 服务器端应用程序

Table of Contents

这部分在 `contrib` 中 包含PostgreSQL服务器相关 应用程序。它们通常运行在数据库服务器存在的主机上。参阅 Reference III, *PostgreSQL 服务器应用程序*获取 关于核心PostgreSQL发布 部分服务器端应用程序的信息。

pg_archivecleanup

Name

pg_archivecleanup -- clean up PostgreSQL WAL archive files

Synopsis

`pg_archivecleanup` [`_option_` ...] `_archivelocation_` `_oldestkeptwalfile_`

Description

pg_archivecleanup is designed to be used as an `archive_cleanup_command` to clean up WAL file archives when running as a standby server (see Section 25.2). pg_archivecleanup can also be used as a standalone program to clean WAL file archives.

To configure a standby server to use pg_archivecleanup, put this into its `recovery.conf` configuration file:

```
archive_cleanup_command = 'pg_archivecleanup _archivelocation_ %r'
```

where `_archivelocation_` is the directory from which WAL segment files should be removed.

When used within archive_cleanup_command, all WAL files logically preceding the value of the `%r` argument will be removed from `_archivelocation_`. This minimizes the number of files that need to be retained, while preserving crash-restart capability. Use of this parameter is appropriate if the `_archivelocation_` is a transient staging area for this particular standby server, but *not* when the `_archivelocation_` is intended as a long-term WAL archive area, or when multiple standby servers are recovering from the same archive location.

When used as a standalone program all WAL files logically preceding the `_oldestkeptwalfile_` will be removed from `_archivelocation_`. In this mode, if you specify a `.backup` file name, then only the file prefix will be used as the `_oldestkeptwalfile_`. This allows you to remove all WAL files archived prior to a specific base backup without error. For example, the following example will remove all files older than WAL file name `000000010000003700000010`:

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg*archivecleanup assumes that `_archivelocation`* is a directory readable and writable by the server-owning user.

Options

pg_archivecleanup accepts the following command-line arguments:

`-d`

Print lots of debug logging output on `stderr`.

`-n`

Print the names of the files that would have been removed on `stdout` (performs a dry run).

`-V``--version`

Print the pg_archivecleanup version and exit.

`-x` `_extension_`

When using the program as a standalone utility, provide an extension that will be stripped from all file names before deciding if they should be deleted. This is typically useful for cleaning up archives that have been compressed during storage, and therefore have had an extension added by the compression program. For example: `-x .gz`.

Note that the `.backup` file name passed to the program should not include the extension.

`-?` `--help`

Show help about pg_archivecleanup command line arguments, and exit.

Notes

pg_archivecleanup is designed to work with PostgreSQL 8.0 and later when used as a standalone utility, or with PostgreSQL 9.0 and later when used as an archive cleanup command.

pg_archivecleanup is written in C and has an easy-to-modify source code, with specifically designated sections to modify for your own needs

Examples

On Linux or Unix systems, you might use:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

where the archive directory is physically located on the standby server, so that the `archive_command` is accessing it across NFS, but the files are local to the standby. This will:

- produce debugging output in `cleanup.log`
- remove no-longer-needed files from the archive directory

Author

Simon Riggs `<[simon@2ndquadrant.com](mailto:simon@2ndquadrant.com)>`

See Also

pg_standby

pg_standby

Name

pg_standby -- supports the creation of a PostgreSQL warm standby server

Synopsis

`pg_standby` [`_option_` ...] `_archivelocation_` `_nextwalfile_` `_xlogfilepath_` [`_restartwalfile_`]

Description

pg_standby supports creation of a "warm standby" database server. It is designed to be a production-ready program, as well as a customizable template should you require specific modifications.

pg_standby is designed to be a waiting `restore_command` , which is needed to turn a standard archive recovery into a warm standby operation. Other configuration is required as well, all of which is described in the main server manual (see Section 25.2).

To configure a standby server to use pg_standby, put this into its `recovery.conf` configuration file:

```
restore_command = 'pg_standby _archiveDir_ %f %p %r'
```

where `_archiveDir_` is the directory from which WAL segment files should be restored.

If `_restartwalfile_` is specified, normally by using the `%r` macro, then all WAL files logically preceding this file will be removed from `_archivelocation_` . This minimizes the number of files that need to be retained, while preserving crash-restart capability. Use of this parameter is appropriate if the `_archivelocation_` is a transient staging area for this particular standby server, but *not* when the `_archivelocation_` is intended as a long-term WAL archive area.

pg*standby assumes that `_archivelocation* `is a directory readable by the server-owning user. If` *restartwalfile* `(or` -k `) is specified, the` *archivelocation*` directory must be writable too.

There are two ways to fail over to a "warm standby" database server when the master server fails:

Smart Failover

In smart failover, the server is brought up after applying all WAL files available in the archive. This results in zero data loss, even if the standby server has fallen behind, but if there is a lot of unapplied WAL it can be a long time before the standby server becomes ready. To trigger a smart failover, create a trigger file containing the word `smart`, or just create it and leave it empty.

Fast Failover

In fast failover, the server is brought up immediately. Any WAL files in the archive that have not yet been applied will be ignored, and all transactions in those files are lost. To trigger a fast failover, create a trigger file and write the word `fast` into it. pg_standby can also be configured to execute a fast failover automatically if no new WAL file appears within a defined interval.

Options

pg_standby accepts the following command-line arguments:

`-c`

Use `cp` or `copy` command to restore WAL files from archive. This is the only supported behavior so this option is useless.

`-d`

Print lots of debug logging output on `stderr`.

`-k`

Remove files from `_archivelocation_` so that no more than this many WAL files before the current one are kept in the archive. Zero (the default) means not to remove any files from `_archivelocation_`. This parameter will be silently ignored if `_restartwalfile_` is specified, since that specification method is more accurate in determining the correct archive cut-off point. Use of this parameter is *deprecated* as of PostgreSQL 8.3; it is safer and more efficient to specify a `_restartwalfile_` parameter. A too small setting could result in removal of files that are still needed for a restart of the standby server, while a too large setting wastes archive space.

`-r` `_maxretries_`

Set the maximum number of times to retry the copy command if it fails (default 3). After each failure, we wait for `_sleeptime_` * `_num_retries_` so that the wait time increases progressively. So by default, we will wait 5 secs, 10 secs, then 15 secs before reporting the failure back to the standby server. This will be interpreted as end of recovery and the standby will come up fully as a result.

`-s` `_sleeptime_`

Set the number of seconds (up to 60, default 5) to sleep between tests to see if the WAL file to be restored is available in the archive yet. The default setting is not necessarily recommended; consult Section 25.2 for discussion.

`-t` `_triggerfile_`

Specify a trigger file whose presence should cause failover. It is recommended that you use a structured file name to avoid confusion as to which server is being triggered when multiple servers exist on the same system; for example `/tmp/pgsql.trigger.5432`.

`-V``--version`

Print the pg_standby version and exit.

`-w` `_maxwaittime_`

Set the maximum number of seconds to wait for the next WAL file, after which a fast failover will be performed. A setting of zero (the default) means wait forever. The default setting is not necessarily recommended; consult Section 25.2 for discussion.

`-?` `--help`

Show help about pg_standby command line arguments, and exit.

Notes

pg_standby is designed to work with PostgreSQL 8.2 and later.

PostgreSQL 8.3 provides the `%r` macro, which is designed to let pg_standby know the last file it needs to keep. With PostgreSQL 8.2, the `-k` option must be used if archive cleanup is required. This option remains available in 8.3, but its use is deprecated.

PostgreSQL 8.4 provides the `recovery_end_command` option. Without this option a leftover trigger file can be hazardous.

pg_standby is written in C and has an easy-to-modify source code, with specifically designated sections to modify for your own needs

Examples

On Linux or Unix systems, you might use:

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r 2>>

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

where the archive directory is physically located on the standby server, so that the `archive_command` is accessing it across NFS, but the files are local to the standby (enabling use of `ln`). This will:

- produce debugging output in `standby.log`
- sleep for 2 seconds between checks for next WAL file availability
- stop waiting only when a trigger file called `/tmp/pgsql.trigger.5442` appears, and perform failover according to its content
- remove the trigger file when recovery ends
- remove no-longer-needed files from the archive directory

On Windows, you might use:

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r 2>>st

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

Note that backslashes need to be doubled in the `archive_command`, but *not* in the `restore_command` or `recovery_end_command`. This will:

- use the `copy` command to restore WAL files from archive
- produce debugging output in `standby.log`
- sleep for 5 seconds between checks for next WAL file availability
- stop waiting only when a trigger file called `C:\pgsql.trigger.5442` appears, and perform failover according to its content
- remove the trigger file when recovery ends
- remove no-longer-needed files from the archive directory

The `copy` command on Windows sets the final file size before the file is completely copied, which would ordinarily confuse pg_standby. Therefore pg_standby waits `sleeptime` seconds once it sees the proper file size. GNUWin32's `cp` sets the file size only after the file copy is complete.

Since the Windows example uses `copy` at both ends, either or both servers might be accessing the archive directory across the network.

Author

Simon Riggs `<[simon@2ndquadrant.com](mailto:simon@2ndquadrant.com)>`

See Also

pg_archivecleanup

pg_test_fsync

Name

pg_test_fsync -- determine fastest wal_sync_method for PostgreSQL

Synopsis

`pg_test_fsync` [`_option_` ...]

Description

pg_test_fsync is intended to give you a reasonable idea of what the fastest wal_sync_method is on your specific system, as well as supplying diagnostic information in the event of an identified I/O problem. However, differences shown by pg_test_fsync might not make any significant difference in real database throughput, especially since many database servers are not speed-limited by their transaction logs. pg_test_fsync reports average file sync operation time in microseconds for each `wal_sync_method`, which can also be used to inform efforts to optimize the value of commit_delay.

Options

pg_test_fsync accepts the following command-line options:

`-f``--filename`

Specifies the file name to write test data in. This file should be in the same file system that the `pg_xlog` directory is or will be placed in. (`pg_xlog` contains the WAL files.) The default is `pg_test_fsync.out` in the current directory.

`-s` `--secs-per-test`

Specifies the number of seconds for each test. The more time per test, the greater the test's accuracy, but the longer it takes to run. The default is 5 seconds, which allows the program to complete in under 2 minutes.

`-V` `--version`

Print the pg_test_fsync version and exit.

`-?` `--help`

Show help about pg_test_fsync command line arguments, and exit.

Author

Bruce Momjian `<[bruce@momjian.us](mailto:bruce@momjian.us)>`

See Also

postgres

pg_test_timing

Name

pg_test_timing -- measure timing overhead

Synopsis

`pg_test_timing` [`_option_` ...]

Description

pg_test_timing is a tool to measure the timing overhead on your system and confirm that the system time never moves backwards. Systems that are slow to collect timing data can give less accurate `EXPLAIN ANALYZE` results.

Options

pg_test_timing accepts the following command-line options:

`-d` `_duration_``--duration=``_duration_`

Specifies the test duration, in seconds. Longer durations give slightly better accuracy, and are more likely to discover problems with the system clock moving backwards. The default test duration is 3 seconds.

`-V` `--version`

Print the pg_test_timing version and exit.

`-?` `--help`

Show help about pg_test_timing command line arguments, and exit.

Usage

Interpreting results

Good results will show most (>90%) individual timing calls take less than one microsecond. Average per loop overhead will be even lower, below 100 nanoseconds. This example from an Intel i7-860 system using a TSC clock source shows excellent performance:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 nsec
Histogram of timing durations:
< usec   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Note that different units are used for the per loop time than the histogram. The loop can have resolution within a few nanoseconds (nsec), while the individual timing calls can only resolve down to one microsecond (usec).

Measuring executor timing overhead

When the query executor is running a statement using `EXPLAIN ANALYZE`, individual operations are timed as well as showing a summary. The overhead of your system can be checked by counting rows with the psql program:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

The i7-860 system measured runs the count query in 9.8 ms while the `EXPLAIN ANALYZE` version takes 16.6 ms, each processing just over 100,000 rows. That 6.8 ms difference means the timing overhead per row is 68 ns, about twice what pg_test_timing estimated it would be. Even that relatively small amount of overhead is making the fully timed count statement take almost 70% longer. On more substantial queries, the timing overhead would be less problematic.

Changing time sources

On some newer Linux systems, it's possible to change the clock source used to collect timing data at any time. A second example shows the slowdown possible from switching to the slower acpi_pm time source, on the same system used for the fast results above:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 nsec
Histogram of timing durations:
< usec   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

In this configuration, the sample `EXPLAIN ANALYZE` above takes 115.9 ms. That's 1061 nsec of timing overhead, again a small multiple of what's measured directly by this utility. That much timing overhead means the actual query itself is only taking a tiny fraction of the accounted for time, most of it is being consumed in overhead instead. In this configuration, any `EXPLAIN ANALYZE` totals involving many timed operations would be inflated significantly by timing overhead.

FreeBSD also allows changing the time source on the fly, and it logs information about the timer selected during boot:

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

Other systems may only allow setting the time source on boot. On older Linux systems the "clock" kernel setting is the only way to make this sort of change. And even on some more recent ones, the only option you'll see for a clock source is "jiffies". Jiffies are the older Linux software clock implementation, which can have good resolution when it's backed by fast enough timing hardware, as in this example:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
< usec   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Clock hardware and timing accuracy

Collecting accurate timing information is normally done on computers using hardware clocks with various levels of accuracy. With some hardware the operating systems can pass the system clock time almost directly to programs. A system clock can also be derived from a chip that simply provides timing interrupts, periodic ticks at some known time interval. In either case, operating system kernels provide a clock source that hides these details. But the accuracy of that clock source and how quickly it can return results varies based on the underlying hardware.

Inaccurate time keeping can result in system instability. Test any change to the clock source very carefully. Operating system defaults are sometimes made to favor reliability over best accuracy. And if you are using a virtual machine, look into the recommended time sources compatible with it. Virtual hardware faces additional difficulties when emulating timers, and there are often per operating system settings suggested by vendors.

The Time Stamp Counter (TSC) clock source is the most accurate one available on current generation CPUs. It's the preferred way to track the system time when it's supported by the operating system and the TSC clock is reliable. There are several ways that TSC can fail to provide an accurate timing source, making it unreliable. Older systems can have a TSC clock that varies based on the CPU temperature, making it unusable for timing. Trying to use TSC on some older multicore CPUs can give a reported time that's inconsistent among multiple cores. This can result in the time going backwards, a problem this program checks for. And even the newest systems can fail to provide accurate TSC timing with very aggressive power saving configurations.

Newer operating systems may check for the known TSC problems and switch to a slower, more stable clock source when they are seen. If your system supports TSC time but doesn't default to that, it may be disabled for a good reason. And some operating systems may not detect all the possible problems correctly, or will allow using TSC even in situations where it's known to be inaccurate.

The High Precision Event Timer (HPET) is the preferred timer on systems where it's available and TSC is not accurate. The timer chip itself is programmable to allow up to 100 nanosecond resolution, but you may not see that much accuracy in your system clock.

Advanced Configuration and Power Interface (ACPI) provides a Power Management (PM) Timer, which Linux refers to as the acpi_pm. The clock derived from acpi_pm will at best provide 300 nanosecond resolution.

Timers used on older PC hardware include the 8254 Programmable Interval Timer (PIT), the real-time clock (RTC), the Advanced Programmable Interrupt Controller (APIC) timer, and the Cyclone timer. These timers aim for millisecond resolution.

Author

Ants Aasma `<[ants.aasma@eesti.ee](mailto:ants.aasma@eesti.ee)>`

See Also

EXPLAIN

pg_upgrade

Name

pg_upgrade -- 升级 PostgreSQL 数据库实例

Synopsis

`pg_upgrade` `-b` `_oldbindir_` `-B` `_newbindir_` `-d` `_olddatadir_` `-D` `_newdatadir_` [`_option_` ...]

描述

pg_upgrade (原名称为 pg_migrator) 允许数据在 PostgreSQL 数据文件中升级到PostgreSQL 新的 大版本而不需要数据的备份/还原，通常适用在升级大版本时，例如从8.4.7升级到当前PostgreSQL的新最新版本。 在小版本之间升级往往是不需要的， 例如 从9.0.1升级到9.0.4。

PostgreSQL大版本更新会定期增加新的功能，这往往导致系统表结构布局经常 改变，但是内部的数据的存储格式很少改动。 针对这种情况 pg_upgrade 通过建立新的系统表和简易的利用旧的用户 数据文件高效的完成升级。如果未来一个新的主要版本改变了数据存储 结构这将导致旧数据结构无法读取， pg_upgrade 将不能 完成升级。（社区会试图避免这种情况的发生。）

pg_upgrade能很好的完成新旧实例的二进制兼容，例如包括 32/64二进制的兼容性编译时间设置检查。检查任何一个外部模块的二进制兼容 是很有必要的，然而不能通过 pg_upgrade来检查。

pg_upgrade 支持从8.3.X升级到最近的 PostgreSQL主要发行版本， 包括快照和alpha版本。

选项

pg_upgrade 可以使用下面命令行参数:

`-b` `_old_bindir_``--old-bindir=``_old_bindir_`

旧的数据实例的可执行文件目录; 环境变量 `PGBINOLD`

`-B` `_new_bindir_` `--new-bindir=``_new_bindir_`

新的数据实例的可执行文件目录; 环境变量 `PGBINNEW`

`-c` `--check`

仅检查实例，不做任何数据更新

`-d` `_old_datadir_` `--old-datadir=``_old_datadir_`

旧的群集数据目录;环境变量 `PGDATAOLD`

`-D` `_new_datadir_` `--new-datadir=``_new_datadir_`

新的群集数据目录;环境变量 `PGDATANEW`

`-j` `--jobs`

同时使用的进程或者线程数

`-k` `--link`

使用硬链接替代拷贝文件到新的数据库实例 (在windows使用NTFS的接点)

`-o` `_options_` `--old-options` `_options_`

options to be passed directly to the old `postgres` command

`-O` `_options_` `--new-options` `_options_`

options to be passed directly to the new `postgres` command

`-p` `_old_port_number_` `--old-port=``_old_portnum_`

旧数据库实例的端口;环境变量 `PGPORTOLD`

`-P` `_new_port_number_` `--new-port=``_new_portnum_`

新数据库实例的端口;环境变量 `PGPORTNEW`

`-r` `--retain`

即使在升级成功也保存相关的SQL和日志文件。

`-u` `_user_name_` `--user=``_user_name_`

数据实例的超级管理员名; 环境变量 `PGUSER`

`-v` `--verbose`

启用详细的内部日志信息

`-V` `--version`

打印版本信息，然后退出

`-?` `-h` `--help`

显示当前帮助，然后退出

用法

用 pg_upgrade进行升级的步骤:

1. 选择性的移动实例的安装目录

   如果你用了指定版本的安装目录，例如 `/opt/PostgreSQL/9.1`，你不需要再移动 旧的安装目录。图形化安装工具都是指定安装目录的。

   如果你的安装目录不是版本指定的，例如 `/usr/local/pgsql`，这就有必要 移动当前的安装目录，这样就避免影响了新的 PostgreSQL 的安装。 当 PostgreSQL 数据库服务停止后， 重命令PostgreSQL的安装 目录的操作是安全的;假如旧安装目录是 `/usr/local/pgsql`，你可以用命令:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   来重命令当前目录名。

2. 用源码安装，构建一个新的版本

   用兼容旧的数据库实例 `configure` 选项来构建新的PostgreSQL。在开始升级之前 pg_upgrade 将会检查 `pg_controldata` 来确保所有的设置都兼容。

3. 安装新的二进制文件

   安装新服务器的二进制可执行文件和支持文件。

   用源码安装时，如果你想要在自定义目录安装新服务器，可以使用 `prefix` 变量:

   ```
   gmake prefix=/usr/local/pgsql.new install
   ```

4. **Install pg_upgrade and pg_upgrade_support**

   在新PostgreSQL的安装中，安装 pg_upgrade 二进制文件 and pg_upgrade_support库文件 。

5. 初始化新的数据库实例

   使用 `initdb` 命令初始化新的数据库实例。 并且，要用兼容的旧数据库实例的 `initdb` 选项 。 很多预构建安装会自动完成这一部。不需要去启动新的数据库实例。

6. 安装自定义的共享对象文件

安装所以旧数据库实例用到的自定义共享对象文件（或者是DLL文件），例如 `pgcrypto.so`，无论他们来自 `contrib` 或者其它的源。 不需要安装一类模式的定义， 例如 `pgcrypto.sql`，因为这些也会从旧数据库实例中升级。

7. 调整连接认证

   `pg_upgrade` 会数次连到新旧数据库实例，所以你修改 `pg_hba.conf`,可以把认证设置成 `trust` 或者是 `peer` ， 另外也可以使用 `md5` 的认证方式， 同时使用 `~/.pgpass` 密码文件 (参考 Section 31.15).

8. 停止新旧数据库

   确保两个数据库都停止使用， 在类Unix操作系统中使用 例如:

```
pg_ctl -D /opt/PostgreSQL/8.4 stop
pg_ctl -D /opt/PostgreSQL/9.0 stop
```

   在Windows中，使用windows可用的服务名：

```
NET STOP postgresql-8.4
NET STOP postgresql-9.0
```

   或者

```
NET STOP pgsql-8.3  (8.3和更早版本的PostgreSQL  使用了不同的服务器)
```

9. 运行 **pg_upgrade**

   运行新数据库的可执行命令 pg_upgrade， 而不是旧数据库的。 pg_upgrade 需要指定新旧数据库实例的数据目录和可执行的 (`bin`) 目录。 当然你还可以指定用户和端口，指定是否使用数据硬链接代替使用 数据复制（默认方式）。

   如果你使用链接模式，升级将会非常快（没有文件拷贝）并且占用更少的硬盘，但是你不能 再访问你的旧数据库当你升级完成启动新的数据库实例。 链接模式还需要新旧数据库 数据目录使用相同的文件系统。 （表空间和 `pg_xlog` 可以在不同的 文件系统。） 参考 `pg_upgrade --help` 查看完整的帮助选项列表。

   `--jobs` 选项可以在拷贝/链接数据文件时使用多CPU核心并且可以并行 的还原数据库模式;一个好的值是CPU核数和表空间的最大值。在一个多核数据库 机器上升级一个多数据库服务器时这个选项能大量的节约时间。

   For Windows users, you must be logged into an administrative account, and 对于Windows用户来说，你必要登录一个管理员的帐号，并且用 `postgres` 用户来启动一个终端设置可用的PATH

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\9.0\bin;
```

然后运行 pg_upgrade 带上引号， 例如:

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/8.4/data"
        --new-datadir "C:/Program Files/PostgreSQL/9.0/data"
        --old-bindir "C:/Program Files/PostgreSQL/8.4/bin"
        --new-bindir "C:/Program Files/PostgreSQL/9.0/bin"
```

开始以后， `pg_upgrade` 会验证两个数据库实例是兼容的 然后开始升级。 你可以使用 `pg_upgrade --check` 去执行检查工作，甚至是旧数据库实例仍在运行。 `pg_upgrade --check` 会概述一些自定义的调整你需要在升级后去查看。如果你使用了链接模式，你必需要 要用 `--link` 参数和 `--check` 来启用指定链接模式的 检查。 `pg_upgrade` 需要对当前目录有可写权限。

在数据库升级过程中不能访问数据库实例这是显而易见的。pg_upgrade 默认在端口50432运行来避免预期之外的数据库连接。当你在升级的时候你可以在 新旧数据库实例上使用同一个端口因为新旧数据库实例不会同时运行 然而， 当查检旧数据库实例时， 新旧数据库实例的端口必需不同。

如果在还原数据库模式时出现错误时， `pg_upgrade` 将会退出你必需参照 step 14的概要信息还原旧实例 再一次使用 `pg_upgrade` ， 你可能需要修改旧实例以让旧数据库模式 升级成功。如果问题出现在某个外部模块上，你可能需要从旧实例上卸载这些外部模块 然后在升级成功后再在新实例上安装它们，假设模块没有被用于储存用户数据。

10. 还原 **`pg_hba.conf`**

    如果你修改 `pg_hba.conf` 成 `trust` 的方式， 还原它的原来的认证设置。 还有可能需要去调整其它的配置文件去和旧的数据库 实例相匹配，例如。 `postgresql.conf` .

11. 升级后的处理

    如果有升级后续操作需要执行，pg_upgrade会在完成后发布出警告信息。 同时它会生成由管理员运行脚本文件。 脚本会连到新旧数据库执行后续操作。 脚本可以用下面命令执行：

    ```
    psql --username postgres --file script.sql postgres
    ```

    脚本可以任意顺序执行，执行完后可以被删除。

    Caution

    通常情况下在重建脚本运行结束之前不允许访问被引用的表;这样做可能会出现 意想不到的错误结果或者是性能不佳 。不引用的表可以被立限访问。

1. 统计

   因为优化统计结果不会被 `pg_upgrade` 传递， 你需要指定去运行命令去在升级 完成后生成一些新的信息。你可能需要设置连接参数去匹配新的数据库实例。

2. 删除旧的数据库实例

   当 `pg_upgrade` 成功的升级完成后，你可能要运行脚本的 方法删除旧的数据库实例的数据目录你可以删除旧安装目录 (例如。 `bin` , `share`)。

3. 还原到旧实例

   如果，运行 `pg_upgrade` 后， 你希望回复到旧实例， 下面是一些选项：

   - 如果你运行了加 `--check` 参数的 `pg_upgrade` ，不会对旧数据实例产生影响可以随时更新实用。
   - 如果你运行了加 `--link` 参数的 `pg_upgrade` 数据文件会被新旧数据实例所共用，如果你启动了新数据实例，新的 数据库会写到那些共享文件上，这对旧实例是不安全的。
   - 如果你运行了不加 `--link` 参数的 `pg_upgrade` 或者没有启动新的数据库，旧实例并没有被修改过，如果链接开始， `.old` 后缀会出现在 `$PGDATA/global/pg_control` 目录下。 为了重新旧数据实例，可以从 `$PGDATA/global/pg_control` 目录下移除 `.old` 后缀的文件;然后你可以重启旧实例。

注意事项

pg_upgrade不支持包含有 `reg*` 类型的OID-引用 比如 以下类型: `regproc` , `regprocedure` , `regoper` , `regoperator` , `regconfig` , and `regdictionary` . (`regtype` 可以被升级。)

所有失败， 重建， 和重建索引会影响到你升级都会被pg_upgrade报告。 升级后的重建表和索引会自动生成。如果你尝试自动升级多个实例，你应该找到 相同的数据模式需要相同的升级后步骤对所以的实例升级;这是因为升级后的步骤 是基于数据库模式的，而不是用户数据。

部署测试，可以创建一个只读的模式从旧数据实例复制，插入一些虚拟的数据， 然后升级他们。

如果你升级PostgreSQL 9.2版本之前只用一个仅配置目录的 的数据库，你必要传递真的数据目录给pg_upgrade,并且给数据库传递配置目录， 例如。 `-d /real-data-directory -o '-D /configuration-directory'` 。

如果用一个9.1之前用了非默认的Unix域socket目录或者默认的位置和新实例 默认位置不一致，要设置 `PGHOST` 来指定旧数据库的socket位置。 (在Windows下不需要。)

一个日志同步的数据库（Section 25.2) 不能被升级原因是 升级数据库必需要允许写。很简单的方式是升级方库然后用rsync去重新建 备份。你可以运行 `rsync` 当主库停用后，或者作为一次基础备份 (Section 24.3.2)去覆盖旧的数据库备份。

如果你想要用link方式并且当新实例启动后你不想要你的旧数据库实例数据被修改 用 `rsync` 从运行中的旧实例新建一个脏的数据复制，然后关闭旧实例 再一次运行 `rsync` 保证所有的数据更改保持一致的。你可能需要 排除一些文件，例如。 `postmaster.pid` , 还有在 Section 24.3.3提到的文件。

从PostgreSQL 8.3升级的局限性

从 PostgreSQL 8.3 升级有一些额外的条件没有被提及 当升级到以后的PostgreSQL发行版本。 举个例子，在8.3版本中如果用户字段中 定义了:

- 一个 `tsquery` 数据类型
- 数据类型 `name` 并不是第一个字段pg_upgrade将不支持升级。

你必需要删除任何这样的字段并且手动升级他们。

如果 `ltree` 模块被安装到数据库中，pg_upgradebu将不支持。

pg_upgrade 会需要重建表如果表：

- 一个字段的数据类型是 `tsvector`

pg_upgrade 会需要重建索引如果：

- 一个索引的类型是hash 或者 GIN
- 一个使用了 `bpchar_pattern_ops` 的索引

同样的，PostgreSQL 8.3之后的版本默认的日期存储格式更改到整型。 pg_upgrade会查检日期存储格式让新旧数据库实例匹配。确保你的新实例是构建时 configure 带上了 `--disable-integer-datetimes` 参数。

对于Windows用户，要注意的不同的整型的日期类型设置是在图形华和MSI安装程序中， 它仅仅可能从版本8.3的安装版到8.4升级或者最近的安装版。不可能从MSI安装程序 到升级新的图形化安装程序中。

参考

initdb, pg_ctl, pg_dump, postgres

pg_xlogdump

Name

pg_xlogdump -- 显示人类易读渲染的PostgreSQL 数据库集合实例的预写日志

Synopsis

`pg_xlogdump` [`option` ...] [`startseg` [`endseg`]]

描述

`pg_xlogdump` 显示预写日志(WAL) 并且主要用于调试或者教学演示。

这个实用工具只能被安装数据库的用户来运行，因为这需要只读的方式访问数据字典。

选项

以下的命令行参数控制了输出的位置和格式：

`_startseg_`

开始从指定WAL段文件读取。这隐式的确定了搜索文件和使用的时间线。

`_endseg_`

指定到哪个WAL段文件停止读取。

`-b``--bkp-details`

输出关于备份数据块的详细信息。

`-e` `_end_` `--end=``_end_`

指定停止读取的日志位置，代替读取的日志流的结尾。

`-n` `_limit_` `--limit=``_limit_`

指定显示记录的条目数。

`-p` `_path_` `--path=``_path_`

指定搜索WAL段文件的目录。默认从当前目录的 `pg_xlog` 子目录去搜索它们。

`-r` `_rmgr_` `--rmgr=``_rmgr_`

过滤由指定资源管理器产生的记录。如果 `list` 一个指定一个名字， 打印一列合法的资源管理器的名字的记录，然后退出。

`-s` `_start_` `--start=``_start_`

开始读取日志的位置。默认是从最早的文件搜索到的第一个有效的日志记录开始。

`-t` `_timeline_` `--timelime=``_timeline_`

从哪个时间线读取日志记录。如果 `startseg` 的值被指定默认就使用startseg; 否则默认值是1。

`-V` `--version`

打印出 pg_xlogdump 版本并且退出。

`-x` `_xid_` `--xid=``_xid_`

仅显示指定事务ID的记录。

`-?` `--help`

显示 pg_xlogdump 命令参数帮助，然后退出。

注意

能在系统运行时得到错误结果。

仅显示指定时间线的记录（如果未指定，就用默认值）。其它时间线的记录将被忽略。

参考

Section 29.5

Appendix H. 外部项目

Table of Contents

PostgreSQL是一项复杂的软件项目，管理它是一项困难的工作。 我们发现许多PostgreSQL的增强 可以通过独立于主项目的方式更好地开发。

H.1. 客户端接口

在基本的PostgreSQL发布中仅包含两个客户端接口：

- libpq被包含的原因是它是主C语言接口， 许多其它客户端接口都依赖于它。
- ECPG被包含的原因是它是它依赖于服务器端SQL语法， 因此对PostgreSQL自身的变化非常敏感。

除此以外的所有其它语言的接口都是外部项目并独立发布，Table H-1 列出了其中的一些。 需要注意的是其中的一些发布许可证与PostgreSQL不同。 要了解更多关于每种语言的接口细节以及许可证等信息，请参考它们各自的文档。

Table H-1. 外部客户端接口

名字	语言	注释	网站
DBD::Pg	Perl	Perl DBI驱动	http://search.cpan.org/dist/DBD-Pg/
JDBC	JDBC	类型4 JDBC驱动	http://jdbc.postgresql.org/
libpqxx	C++	新型C++接口	http://pqxx.org/
Npgsql	.NET	.NET数据提供者	http://npgsql.projects.postgresql.org/
pgtclng	Tcl		http://sourceforge.net/projects/pgtclng/
psqlODBC	ODBC	ODBC驱动	http://psqlodbc.projects.postgresql.org/
psycopg	Python	DB API 2.0-兼容	http://initd.org/psycopg/

H.2. 管理工具

有几个PostgreSQL可用的管理工具。 最受欢迎的是pgAdmin III， 还有一些商业上可用的。

H.3. 过程语言

PostgreSQL在基本发布中包含 PL/pgSQL， PL/Tcl， PL/Perl和PL/Python过程语言。

除此之外，还有一些在基本PostgreSQL发布之外开发和维护的过程语言， Table H-2列出了其中的一些。 需要注意的是其中的一些发布许可证与PostgreSQL不同。 要了解更多关于每种过程语言的细节以及许可证等信息，请参考它们各自的文档。

Table H-2. 外部维护的过程语言

名字	语言	网站
PL/Java	Java	http://pljava.projects.postgresql.org/
PL/PHP	PHP	http://www.commandprompt.com/community/plphp/
PL/Py	Python	http://python.projects.postgresql.org/backend/
PL/R	R	http://www.joeconway.com/plr/
PL/Ruby	Ruby	http://raa.ruby-lang.org/project/pl-ruby/
PL/Scheme	Scheme	http://plscheme.projects.postgresql.org/
PL/sh	Unix shell	http://plsh.projects.postgresql.org/

H.4. 扩展

PostgreSQL很容易设计成可以扩展的。 因此，加载到数据库的扩展可以像编译特性一样。 `contrib/` 目录附带包含若干扩展的源代码。 在Appendix F中被描述。其它扩展是独立开发的， 比如PostGIS。 甚至PostgreSQL的复制方案也是在外部开发的。 比如 Slony-I 是一个流行的主/从复制方案，它就是独立在核心项目之外开发的。

Appendix I. 源代码库

PostgreSQL 的源代码存放和管理是 通过 Git 版本控制系统。主源代码库 的公共镜像是可用的; 它会在一分钟内从主源代码库同步所有更新。

在我们的wiki, http://wiki.postgresql.org/wiki/Working_with_Git， 上面有一些关于使用Git的使用说明。

注意： 从源代码库构建PostgreSQL 需要合理 最新版本的 bison， flex， 和Perl工具。 当从版本发行包构建PostgreSQL 并不需要这些工具， 因为这些文件已经包含在发行包里面。其它工具需求和以下显示一样 Chapter 15。

I.1. 获得源代码通过Git

使用 Git 你可以从源代码库复制一份完整源代码到你的本机上， 这样你就可以离线访问所有的代码历史和分支。这是开发或者测试补丁最为快捷并且灵活的方式。

Git

1. 你将需要一个已经安装的Git， 你可以从 http://git-scm.com得到。 大多数操作系统已经默认安 装有最新版的 Git ， 或者在软件包管理系统里提供了下载安装。
2. 为了开始使用Git源码库， 从官方的镜像里制作一份克隆：

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```

   这将会完全拷贝源到你的本地机器上， 所以直到拷贝完成会花费一段时间， 尤其当你的网络连接非常慢时候。 代码文件会放置在当前目录一个新的名为 `postgresql` 的子目录里。

   Git的镜像同样可以通过HTTP协议来获取， 假如你的防火墙阻止你通过Git协议 访问。 仅需要修改URL的协议为 `http` ， 例如：

   ```
   git clone http://git.postgresql.org/git/postgresql.git
   ```

   HTTP协议是效率较低的相比于Git协议， 所以使用起来会更慢。
3. 每当你想在系统得到最近更新时， `cd` 到源目录， 运行：

   ```
   git fetch
   ```

Git 不仅仅获取资源，还能做很多其它的事情。更多详情， 参考 Git 手册页， 或者查看 网站 http://git-scm.com。

Appendix J. 文档

Table of Contents

PostgreSQL有四种主要的文档格式：

- 纯文本，用于安装前信息的纯文本格式。
- HTML，用于在线浏览和参考。
- PDF或PostScript，用于打印。
- 手册页，用于快速参考。

另外，在PostgreSQL源码树里面还有许多 `README` 风格的文件，它们记录了许多与实现有关的东西。

HTML文档和手册页(man pages)是标准版本的一部分并且缺省时安装。 PDF和PostScript格式的文档可以独立地下载。

J.1. DocBook

文档源文件是用*DocBook*写的，它是一种标记语言， 类似HTML。这两种语言都是SGML (标准通用标记语言)的应用，它实际上是描述另外一种语言的语言。 也就是说，术语上DocBook 和SGML都可以用，而技术上是不能互换的。

DocBook允许作者不用考虑表现细节地描述一份技术文档的结构和内容。 一份文档风格定义该内容如何呈现为几种最终形式之一。DocBook 是由 OASIS 工作组组维护的。 DocBook官方网站 有很好的介绍和参考手册以及一整本 O'Reilly 的书可供你在线阅读。 NewbieDoc Docbook Guide非常适合初学者阅读。 FreeBSD Documentation Project也使用 DocBook 并且有一些很好的信息， 包括一些很值得考虑的风格向导。

J.2. 工具集

下面的工具用于处理此文档。有些可能是可选的，在文中标注了。

DocBook DTD

这是 DocBook 本身的定义。目前使用版本 4.2 ；你不能使用更新或者早些的版本。 你需要SGML版本的DocBook DTD，但是建立手册页也需要 XML版本的相同版本。

ISO 8879 character entities

这是 DocBook 需要的，但是独立发布，因为它们是由 ISO 维护的。

DocBook DSSSL Stylesheets

这些东西包含把 DocBook 源代码转换成其它格式(比如HTML)的处理指令。

DocBook XSL Stylesheets

这是另外一个转化DocBook到其他格式的样式表。我们当前使用它来生成手册页和可选的HTML帮助。 你也可以使用这个工具生成HTML或PDF输出，但是官方的PostgresSQL发布使用DSSSL stylesheets。

当前所需的最小版本是1.74.0。

OpenJade

这是处理SGML的基本包。它包含一个SGML分析器， 一个DSSSL处理器(也就是一个用DSSSL风格表把 SGML转换成其它格式的程序)， 还有一系列相关工具。现在Jade 由 OpenJade 组维护，而不再是 James Clark 了。

Libxslt for `xsltproc`

这是和XSLT stylesheets一起使用的处理工具（类似 `jade` ，是处理DSSSL stylesheets的工具）。

JadeTeX

如果你需要，你还可以安装JadeTeX把TeX 用做Jade的一种格式化后端。JadeTeX 可以生成 Postscript 或者PDF文件(后者带书签)。

不过，JadeTeX的输出比RTF后端稍差一点的打印输出。 主要是表的格式和各种竖直和水平的空白效果。而且，你还没有机会手工润色输出结果。

我们已经在文档中记录了几种安装处理此文档所需的各种工具的方法。它们在下面描述。 也可能有其它包发布这些工具。请向 doc 邮件列表报告那些包的状态，就会在这里包括那些信息。

J.2.1. Linux RPM 安装

许多供应商在它们的版本里提供了一整套处理 DocBook 的 RPM 包，请检查一下"SGML"选项， 或者下列包之一：`sgml-common`，`docbook`，`stylesheets`，`openjade`(或`jade`)。 可能还需要`sgml-tools`和`xsltproc`或`libxslt`。 如果你的版本没有提供这些东西，那么你应该可以使用来自一些其它合理兼容的发行商的包。

J.2.2. FreeBSD 安装

FreeBSD 文档计划本身就非常频繁地使用 DocBook， 所以在 FreeBSD 里有一整套可以用的文档工具的"ports"就一点也不奇怪了。 要在 FreeBSD 里制作文档，你必须安装下面的 port ：

- `textproc/sp`
- `textproc/openjade`
- `textproc/iso8879`
- `textproc/dsssl-docbook-modular`
- `textproc/docbook-420`

很多东西来自`/usr/ports/print`(`tex`， `jadetex`)你也可能会安装。

这些 port 很可能不会更新位于`/usr/local/share/sgml/catalog.ports`的主目录文件或顺序不合适。确保文件的开头有下面这几行：

```
CATALOG "openjade/catalog"
CATALOG "iso8879/catalog"
CATALOG "docbook/dsssl/modular/catalog"
CATALOG "docbook/4.2/catalog"
```

如果你不想编辑文件，还可以把环境变量`SGML_CATALOG_FILES`设置为一个冒号分隔的目录文件列表(比如上面那样的)。

你可以在FreeBSD Documentation Project's instructions里找到更多有关 FreeBSD 文档工具的信息。

J.2.3. Debian 包

Debian GNU/Linux里面也有一整套可以用的文档工具的包。 安装时，只需要用下面的命令：

```
apt-get install docbook docbook-dsssl docbook-xsl openjade1.3 opensp xsltproc
```

J.2.4. Mac OS X

如果你使用MacPorts，你将使用以下命令来设置：

```
sudo port install docbook-dsssl docbook-sgml-4.2 docbook-xml-4.2 docbook-xsl libxslt open
```

J.2.5. 从源程序手工安装

DocBook 工具的手工安装过程有些复杂，因此如果你有预先制作好的包，最好还是用它们。在这里只描述一个标准的安装，而且安装到标准的路径里，并且没有"神奇"的特性。 相关的更多的细节，你应该学习相关包的文档，并且阅读SGML介绍性材料。

J.2.5.1. 安装 OpenJade

1. OpenJade 提供了一个 GNU 风格的 `./configure; make; make install` 制作过程。 你可以在 OpenJade 源程序包里找到详细内容。在 shell 里：

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   make
   make install
   ```

   确保你记住了"default catalog"的位置，后面还会需要它。也可以不用注明这句话， 但是稍后使用jade的时候，你就不得不把环境变量 `SGML_CATALOG_FILES` 设置为指向该文件的位置。如果 OpenJade 已经安装，并且你想在本地安装其它工具的时候，这也是个可选的方法。

   > **Note:** 一些用户已经报告了使用OpenJade 1.4devel建立PDF时的分段错误，信息如下：> > `> openjade:./stylesheet.dsl:664:2:E: flow object not accepted by port; only display` > > 使用低级的OpenJade 1.3应该鞥解决这个错误。

2. 另外，你应该从 `dsssl` 目录里把 `dsssl.dtd`，`fot.dtd`，`style-sheet.dtd`，`catalog` 文件安装上，可能是安装到 `/usr/local/share/sgml/dsssl` 吧。 最简单的可能就是复制整个目录：

   ```
   cp -R dsssl /usr/local/share/sgml
   ```

3. 最后，创建文件 `/usr/local/share/sgml/catalog` 并且把下面行加入其中：

   ```
   CATALOG "dsssl/catalog"
   ```

这是一个相对路径，指向在step 2里安装的文件。 请根据你自己的安装布局进行调整。

J.2.5.2. 安装 DocBook DTD 工具箱

1. 获取DocBook V4.2 发布

2. 创建目录 `/usr/local/share/sgml/docbook-4.2` 并且进入该目录。 实际的位置并非关键，上面这个只是在这里的布局的比较合理的位置。

```
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;mkdir /usr/loca
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;cd /usr/local/s
```

3. 解包归档：

```
&lt;samp class="literal"&gt;$&lt;/samp&gt; &lt;kbd class="literal"&gt;unzip -a ...../
```

   这个归档将把它的文件解开到当前目录。

4. 编辑 `/usr/local/share/sgml/catalog` 文件(或者任何安装的时候你告诉 jade 的东西) 并且把类似下面的行放到该文件里面：

```
CATALOG "docbook-4.2/docbook.cat"
```

5. 下载ISO 8879 字符记录归档， 解开它，然后把文件放到 DocBook 文件的同一个目录里。

6. 在装有 DocBook 和 ISO 文件的目录里运行下面的命令：

```
perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
```

   这个动作修补了一个小毛病，这个毛病把 DocBook 目录文件里使用的名字和 ISO 字符实体文件的名字混淆了。

J.2.5.3. 安装 DocBook 的DSSSL样式表

要安装样式表，解开发布的工具包，然后把它挪到一个合适的地方(比如 `/usr/local/share/sgml`)。 归档会自动生成一个子目录。

```
<samp class="literal">$</samp> <kbd class="literal">gunzip docbook-dsssl-1.`_xx_`.tar.gz<
<samp class="literal">$</samp> <kbd class="literal">tar -C /usr/local/share/sgml -xf docb
```

你也可以在 `/usr/local/share/sgml/catalog` 里制作常用的目录条目：

```
CATALOG "docbook-dsssl-1._xx_/catalog"
```

因为样式表变化频繁，因此有时候多实验几个版本也挺好，PostgreSQL 并不使用这个表项。参阅Section J.2.6获取有关如何选择样式表的信息。

J.2.5.4. 安装 JadeTeX

要安装和使用JadeTeX，就需要一套能用的TeX 和LaTeX2e，包括支持的工具和 图形包。Babel、 AMS 字体 、AMS-LaTeX 、 PSNFSS扩展、"the 35 fonts"工具箱、 用于生成PostScript 的dvips程序， 宏包fancyhdr。 hyperref， minitoc， url， ot2enc，所有这些你都可以在你最近的 CTAN 镜像站点找到。 基本TeX系统的安装远远超出了这份介绍的范围。 你应该可以在任何可以运行TeX的系统上找到二进制包。

在你开始使用JadeTeX处理PostgreSQL文档之前， 你需要增大TeX的内部数据结构的尺寸。关于这些事情的细节可以在JadeTeX的安装指导里找到。

一旦完成了这些你就可以安装JadeTeX了：

```
<samp class="literal">$</samp> <kbd class="literal">gunzip jadetex-`_xxx_`.tar.gz</kbd>
<samp class="literal">$</samp> <kbd class="literal">tar xf jadetex-`_xxx_`.tar</kbd>
<samp class="literal">$</samp> <kbd class="literal">cd jadetex</kbd>
<samp class="literal">$</samp> <kbd class="literal">make install</kbd>
<samp class="literal">$</samp> <kbd class="literal">mktexlsr</kbd>
```

最后两步需要以root身份处理。

J.2.6. `configure` 检测

在你制作文档之前，你需要像制作程序本身那样运行 `configure` 脚本。 检查运行结尾处的输出， 应该看起来像这样：

```
<samp class="literal">checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/share/sgml/docbook/stylesheet/dsssl/modular
checking for collateindex.pl... /usr/bin/collateindex.pl
checking for xsltproc... xsltproc
checking for osx... osx</samp>
```

如果 `onsgmls` 和 `nsgmls` 都没有找到， 那么下面的测试将被跳过。 `nsgmls` 是 Jade 包的一部分。 可以通过传递环境变量 `JADE` 和 `NSGMLS` 给 configure 来指定这些程序的位置。如果没有找到"DocBook V4.2"， 那么就是你没有把 DocBook DTD 工具箱装到 jade 可以找到的地方，

或者你没有正确设置目录文件。 参阅上面的安装提示。配置脚本会在一些比较标准的位置寻找 DocBook 样式表， 但如果你把它们放在其它位置，那么就应该设置环境变量 `DOCBOOKSTYLE` 为该位置并且重新运行`configure`脚本。

J.3. 制作文档

一旦把所有的东西都设置好了，进入目录 `doc/src/sgml` 然后运行下面其中一条命令(记得要用GNU make)：

J.3.1. HTML

制作HTML版本的文档：

```
<samp class="literal">doc/src/sgml$</samp> <kbd class="literal">gmake html</kbd>
```

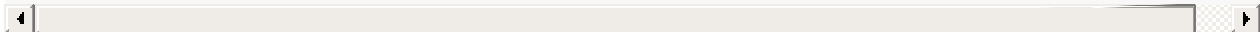
这也是缺省目标。在子目录 `html` 里输出。

要创建一个适当的索引，可能处理几个相同的阶段。如果你不关心该索引， 只是想校对输出，使用 `draft`：

```
<samp class="literal">doc/src/sgml$</samp> <kbd class="literal">gmake draft</kbd>
```

要建立文档为一个HTML页，使用：

```
<samp class="literal">doc/src/sgml$</samp> <kbd class="literal">gmake postgres.html</kbd>
```



J.3.2. 手册页

用DocBook XSL样式表把DocBook的 `refentry` 页面转换成适于做手册页的 *roff 输出。这些手册页也是以 tar 归档的形式发布的， 与HTML版本类似。要创建手册页包，用命令：

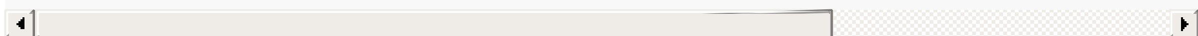
```
cd doc/src/sgml
gmake man
```

J.3.3. 用JadeTeX生成的打印输出

如果你想用JadeTex生成一个可打印的文档，可以用下面的命令：

- 使用DVI生成一个A4格式的PostScript：

  ```
  &lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;gma
  ```

U.S.信件的格式：

```
&lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;gma
```

- 制作一个PDF：

```
&lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;gma
```

或：

```
&lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;gma
```

当然，你也可以从 Postscript 里制作PDF版本，但是如果你直接生成 PDF，那么它会有超链接和其它增强的特性。

当使用 JadeTeX 生成PostgreSQL文档时，你可能需要增大一些TeX的内部参数。 通过设置文件 `texmf.cnf` 来实现。下面的设置立即生效：

```
hash_extra.jadetex  = 200000
hash_extra.pdfjadetex  = 200000
pool_size.jadetex = 2000000
pool_size.pdfjadetex = 2000000
string_vacancies.jadetex = 150000
string_vacancies.pdfjadetex = 150000
max_strings.jadetex = 300000
max_strings.pdfjadetex = 300000
save_size.jadetex = 15000
save_size.pdfjadetex = 15000
```

J.3.4. 溢出文本

偶尔文本超出了打印的边缘，并且在极限情况下，超出了打印的纸张的宽度，例如，非包裹的文本， 宽表格。过宽的文本在TeX日志输出文件中产生"Overfull hbox"信息，例如， `postgres-US.log` 或 `postgres-A4.log` 。一英寸有72个点， 所以任何超过72个点的报告都太宽，可能不适合打印页（假设一英寸）。 要找到导致溢出的SGML文本，首先找到上面提到的溢出信息的首页码，例如， `[50 ###]` (page 50)，然后在PDF文件中查找其后的页码(例如页 51)， 查看溢出文本并相应的调整SGML。

J.3.5. 通过RTF生成打印输出

你也可以通过把它转换成RTF并且用一个办公套件进行格式微调的办法来创建一个PostgreSQL文档的可打印的版本。根据你使用的不同的办公套件， 然后你就可以分别把文档转换成 PDF格式的Postscript。 下面的步骤演示了使用Applixware实现的过程。

> **Note:** 目前看来PostgreSQL的当前版本的文档碰到了 OpenJade 的大小限制的一些毛病。 如果制作RTF版本的时候停住了好长时间，而输出文件还是 0， 那么你很有可能碰到了这个毛病。不过，正常的制作要花 5 到 10 分钟，因此不要太快退出。

Applixware RTF 清理

OpenJade忽略了声明文本主体的缺省风格。以前， 这个未经查明的问题导致目录生成的长时间处理。不过，在Applixware 的工作人员的全力帮助下，这个病症被诊断出来并且找到了绕开的办法。

1. 键入下面命令生成RTF版本：

```
&lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;gma
```

2. 修复 RTF 文件，以正确声明所有风格，尤其是缺省风格。如果文档包含 `refentry` 段，那么还必须把和前面的段落与当前段落绑定的格式化暗示替换为当前的段落和后面的段落绑定。 在 `doc/src/sgml` 里有一个 `fixrtf` 用于完成这样的修补：

```
&lt;samp class="literal"&gt;doc/src/sgml$&lt;/samp&gt; &lt;kbd class="literal"&gt;./f
```

   该脚本把 `{\s0 Normal;}` 增加为文档的零级风格。根据 Applixware，RTF 标准会禁止增加一种隐含的零级风格， 尽管 Microsoft Word 碰巧可以处理这种情况。为了修复 `refentry` 段落， 该脚本把 `\keepn` 标记替换为 `\keep` 。

3. 在Applixware Words里打开新的文档，然后输入该RTF文件。

4. 用Applixware生成一个新的目录(ToC)。

   i. 选择现有的 ToC 行，从第一行第一个字符到最后一行最后一个字符。

   ii. 用Tools->Book Building->制作一个新的 ToC。选择头三层头用于包含在 ToC 里。这将用本地的 Applixware ToC 代替从 RTF 里输入进来的行。

   iii. 使用Format->Style 调整 ToC 格式，选择每三种 ToC 风格，然后为 `First` 和 `Left` 调整边距。使用下面的值：

风格	第一边距(英寸)	左边距(英寸)
`TOC-Heading 1`	`0.4`	`0.4`
`TOC-Heading 2`	`0.8`	`0.8`
`TOC-Heading 3`	`1.2`	`1.2`

5. 对文档进行加工：
    - 调整分页符
    - 调整表列宽
6. 用正确的值替换 ToC 里例子和图片部分右对齐的页数。这些对每个文档只需要花几分钟。
7. 如果索引是空的，那么从文档中删除它。
8. 重新生成并调整目录。
    i. 选择 ToC 字段。
    ii. 选择Tools->Book Building->Create Table of Contents。
    iii. 通过选择Tools->Field Editing->Unprotect解除 ToC。
    iv. 删除 ToC 中的第一行，它是指向 ToC 本身的一条记录。
9. 把该文档保存为Applixware Words本地文档格式以便于最后的编辑。
10. 把该文档以 PostScript 格式"打印"到一个文件。

J.3.6. 纯文本文件

有好几个文件是以纯文本的模式发布的，主要是为了在安装过程中阅读。 `INSTALL` 文件对应 Chapter 15，只有一点用于不同环境的修改。要重建这个文件， 进入目录 `doc/src/sgml` 然后敲入<kbd class="literal">gmake INSTALL</kbd>。 这样就会创建一个叫 `INSTALL.html` 的文件，你可以用 Netscape Navigator把它另存为一个文本文件，然后把它拷到现存文件的位置。 好像Netscape提供了最高的HTML 到文本的转换质量(比lynx和w3m好)。

文件 `HISTORY` 可以用类似方法创建，用的命令是<kbd class="literal">gmake HISTORY</kbd>。 对于 `src/test/regress/README` 文件，命令是<kbd class="literal">gmake regress_README</kbd>。

J.3.7. 语法检查

制作文档可能需要很长时间。但是有一个方法用于只检查文档文件的语法正确性，只要花几秒钟：

```
<samp class="literal">doc/src/sgml$</samp> <kbd class="literal">gmake check</kbd>
```

J.4. 文档写作

SGML和DocBook没有受到过多的开放源码写作工具的影响。 最常用的工具集是带有合适编辑模式的Emacs/XEmacs 编辑器。在一些系统上这些工具在典型的完全安装时是一并安装的。

J.4.1. Emacs/PSGML

PSGML是最常用和最强大的编辑SGML文档的工具。 如果正确的做了配置，它将允许你使用Emacs插入标签和检查标记一致性。 你也可以把它用于HTML。看看PSGML 站点获取下载、安装指导、详细文档。

关于PSGML有一件比较重要的事情要注意：它的作者假设你的主 SGML DTD目录是 `/usr/local/lib/sgml` 。 如果你像本文的例子那样放在 `/usr/local/share/sgml` ，就得补偿这个问题， 要么是设置 `SGML_CATALOG_FILES` 环境变量，要么是自定义你的PSGML 安装(它的手册告诉你怎么做)。

把下面这几行放到你的 `~/.emacs` 环境文件里(根据你的系统调整路径名)：

```
; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

并且在同一个文件里增加一条记录，把SGML 加入自动模式别名的定义中(`auto-mode-alist`)：

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

当使用PSGML时，有一个让自己在这些分离的文件上干活方便些的办法： 就是你在编辑它们的时候插入合适的 `DOCTYPE` 定义。例如， 如果你在这个源文件上干活，这是一个附录章节，因此你将通过把第一行标记成像下面的样子从而把这个文档声明为一个 DocBook 文档的"附录"：

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

这意味着任何或所有读取SGML的软件将能正确读取这份文件， 并且我可以用 `nsgmls -s docguide.sgml` 校验此文档 (不过你在制作整个文档集的时候要把这行拿走)。

J.4.2. 其它 Emacs 模式

GNU Emacs带有不同的SGML模式， 不过并不像PSGML那么强大，但是它比较少让人混淆的东西而且比较小巧。 同样，它也提供语法高亮(字体锁)，也是很有帮助的。 `src/tools/editors/emacs.samples` 包含这个模式的相同设置。

Norm Walsh 提供一个用于 DocBook 的 major mode， 也有字体锁和一些可以减少击键的特性。

J.5. 风格指导

J.5.1. 参考页

参考页应该遵循标准的布局。这样就允许用户更快地找出自己需要的信息， 并且也可以鼓励作者把一条命令的所有相关方面都记录在案。一致性不仅仅是 PostgreSQL参考页的需求，也是操作系统和其它页面提供的东西。 因此设计了下面的指导方针。它们在大多数时候是与各种操作系统建立起来的类似的风格是一致的。

描述可执行命令的参考页应该包含下面的小节，并且是按照这里的顺序。 不适用的小节可以忽略。额外的顶级小节应该只用在特殊的环境下； 通常那些信息属于"用法"小节。

名字（Name）

这个小节是自动生成的。它包含命令名和一个半句话的摘要，介绍该命令的功能。

纲要（Synopsis）

这个小节包含该命令的语法图表。大纲通常不应该列出每个命令行选项；那些东西在后面做。 大纲应该列出命令行的主要组件，比如应该把输入和输出文件放在哪里等。

描述（Description）

几个段落，用于描述该命令是做什么的。

选项（Options）

一个列表，描述每个命令行选项。如果有许多选项，可以用子小节。

退出状态(Exit Status)

如果程序用 0 表示成功，非零表示失败，那么你不需要为此写文档。 如果在每个非零值的后面有不同的含义，那么在这里列出它们。

用法(Usage)

描述任意子语言或者程序的运行时接口。如果程序不是交互的，那么本节通常可以省略。 否则，本节是用于描述所有运行时特性的地方。如果合适，可以使用子小节。

环境(Environment)

列出所有程序可能使用的环境变量。尽量完整；即使是那些看起来很琐碎的变量， 比如 `SHELL` 都可能让读者感兴趣。

文件(Files)

列出所有程序可能隐含访问的文件。也就是说，不要列出在命令行上声明的输入和输出文件。 但是列出配置文件等等。

诊断(Diagnostics)

解释任何程序可能生成的不正常的输出。避免列出所有可能的错误信息。 这样做工作量很大但没有太多实际用途。但是如果说错误信息有一种用户可以分析的标准格式， 那么这里可能就是解释它的地方。

注意(Notes)

任何在其它地方放都不合适的东西，尤其是臭虫、实现缺陷、安全考量、兼容性问题等。

例子(Examples)

例子

历史(History)

如果在程序的历史中有一些主要的里程碑，那么可以在这里列出。通常，这个小节可以省略。

作者（Author）

作者（只在普通发布版中使用）

又见(See Also)

交叉引用，按照下面的顺序列出：其它PostgreSQL 命令的参考页PostgreSQL SQL命令参考页、 PostgreSQL 手册的引用、其它引用页面(比如， 操作系统、其它包)、其它文档。在同一组里的条目按照字母顺序列出。

描述 SQL 命令的参考页应该包含下面的小节：名字、大纲、描述、参数、输出、注意、例子、 兼容性、 历史、又见。参数小节类似选项小节，但是有更多自由可以选择该命令的哪个子句可以列出。 输出小节只在命令有返回的时候需要，而不是缺省的命令结束标签。 兼容性小节应该解释此命令遵循 SQL 标准的哪个扩展，或者它兼容哪种其它数据库系统。 SQL 命令的"又见"小节应该在交叉引用其它程序之前列出 SQL 命令。

Appendix K. 首字母缩略词

这是一个在PostgreSQL使用说明 和PostgreSQL讨论中经常用到的首字母缩略图。

ANSI

American National Standards Institute

API

Application Programming Interface

ASCII

American Standard Code for Information Interchange

BKI

Backend Interface

CA

Certificate Authority

CIDR

Classless Inter-Domain Routing

CPAN

Comprehensive Perl Archive Network

CRL

Certificate Revocation List

CSV

Comma Separated Values

CTE

Common Table Expression

CVE

Common Vulnerabilities and Exposures

DBA

Database Administrator

DBI

Database Interface (Perl)

DBMS

Database Management System

DDL

Data Definition Language,SQL命令比如 `CREATE TABLE` , `ALTER USER`

DML

Data Manipulation Language, SQL命令比如 `INSERT` , `UPDATE` , `DELETE`

DST

Daylight Saving Time

ECPG

Embedded C for PostgreSQL

ESQL

Embedded SQL

FAQ

Frequently Asked Questions

FSM

Free Space Map

GEQO

Genetic Query Optimizer

GIN

Generalized Inverted Index

GiST

Generalized Search Tree

Git

Git)

GMT

Greenwich Mean Time

GSSAPI

Generic Security Services Application Programming Interface

GUC

Grand Unified Configuration, 处理服务器配置的PostgreSQL子系统。

HBA

Host-Based Authentication

HOT

Heap-Only Tuples

IEC

International Electrotechnical Commission

IEEE

Institute of Electrical and Electronics Engineers

IPC

Inter-Process Communication

ISO

International Organization for Standardization

ISSN

International Standard Serial Number

JDBC

Java Database Connectivity

LDAP

Lightweight Directory Access Protocol

MSVC

Microsoft Visual C

MVCC

Multi-Version Concurrency Control

NLS

National Language Support

ODBC

Open Database Connectivity

OID

Object Identifier

OLAP

Online Analytical Processing

OLTP

Online Transaction Processing

ORDBMS

Object-Relational Database Management System

PAM

Pluggable Authentication Modules

PGSQL

PostgreSQL

PGXS

PostgreSQL Extension System

PID

Process Identifier

PITR

Point-In-Time Recovery (连续归档)

PL

Procedural Languages (server-side)

POSIX

Portable Operating System Interface

RDBMS

Relational Database Management System

RFC

Request For Comments

SGML

Standard Generalized Markup Language

SPI

Server Programming Interface

SP-GiST

Space-Partitioned Generalized Search Tree

SQL

Structured Query Language

SRF

Set-Returning Function

SSH

Secure Shell

SSL

Secure Sockets Layer

SSPI

Security Support Provider Interface

SYSV

Unix System V

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)

TID

Tuple Identifier

TOAST

The Oversized-Attribute Storage Technique

TPC

Transaction Processing Performance Council

URL

Uniform Resource Locator

UTC

Coordinated Universal Time

UTF

Unicode Transformation Format

UTF8

Eight-Bit Unicode Transformation Format

UUID

Universally Unique Identifier

WAL

Write-Ahead Log

XID

Transaction Identifier

XML

Extensible Markup Language

参考书目

选择一些SQL 和PostgreSQL的参考和读物。

一些来自最初的POSTGRES 开发队伍的白皮书和 技术报告可以在加州大学伯克利分校计算机科学学系 网站获取。

SQL参考书

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, 实用*SQL*手册__*: Using SQL Variants*, Fourth Edition, Addison-Wesley Professional, ISBN 0-201-70309-2, 2001.

C. J. Date and Hugh Darwen, *SQL*标准指南__*: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, 数据库系统介绍, Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.

Ramez Elmasri and Shamkant Navathe, 数据库系统原理, Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.

Jim Melton and Alan R. Simon, 理解新的*SQL*__*: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, 数据库知识原理__*: Base Systems*, Volume 1, Computer Science Press, 1988.

PostgreSQL-具体文档

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

讨论SQL历史和语法，并且描述了 `INTERSECT` 和 `EXCEPT` 添加构建到PostgreSQL中。 为硕士论文做准备得到O. Univ. Prof. Dr. Georg Gottlob和维也纳理工大学 Univ. Ass. Mag. Katrin Seyr的支持。

A. Yu and J. Chen, The POSTGRES Group, *Postgres95*用户手册, University of California, Sept. 5, 1995.

Zelaine Fong, *POSTGRES*查询优化器的设计和实现 , University of California, Berkeley, Computer Science Department.

程序和文章

Nels Olson, *POSTGRES中部分索引:研究计划*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, "数据库系统中为版本模型使用生产规则的统一框架", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe and M. Stonebraker, " POSTGRES数据模型 ", Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, "Generalized Partial Indexes (cached version) ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.

M. Stonebraker and L. Rowe, " POSTGRES的设计 ", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, "POSTGRES规则系统的设计", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, " POSTGRES存储系统设计 ", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, " POSTGRES规则系统注解 ", *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, " 部分索引实例 ", *SIGMOD Record 18(4)*, Dec. 1989, 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, " POSTGRES的实现 ", *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, " 数据库系统中的规则，过程，缓存和视图 ", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z